M.Sc. Engg. (CSE) Thesis

A Load Balanced Collaborative Content Caching Strategy in Named Data Network

Submitted by

Mina

0413052108

Supervised by

Dr. Mahmuda Naznin

Submitted to

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Dhaka, Bangladesh

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

September 2019

Candidate's Declaration

I, do, hereby, certify that the work presented in this thesis, titled, "A Load Balanced Collaborative Content Caching Strategy in Named Data Network", is the outcome of the investigation and research carried out by me under the supervision of Dr. Mahmuda Naznin, Professor, Department of CSE, BUET.

I also declare that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Mina

Mina

0413052108

The thesis titled "**A Load Balanced Collaborative Content Caching Strategy in Named Data Network**", submitted by Mina, Student ID 0413052108, Session April 2013, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfilment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents on September 29, 2019.

Board of Examiners

1.	[illegible] Dr. Mahmuda Naznin Professor Department of CSE, BUET, Dhaka	Chairman (Supervisor)
2.	[illegible] Dr. Md. Mostofa Akbar Professor and Head Department of CSE, BUET, Dhaka	Member (Ex-Officio)
3.	[illegible] Dr. Md. Shohrab Hossain Associate Professor Department of CSE, BUET, Dhaka	Member
4.	[illegible] Dr. Rifat Shahriyar Associate Professor Department of CSE, BUET, Dhaka	Member
5.	Lutfa Akter Dr. Lutfa Akter Professor Department of EEE BUET, Dhaka	Member (External)

Acknowledgement

I am thankful to my honorable thesis supervisor, Dr. Mahmuda Naznin for her valuable guidance, kind nurture, colossal effort and mental support. She spared her time to guide and motivated me, held the torch at the time of my darkest phases during this research. Without her guidance, I wouldn't be able to complete my research. I cannot thank her more.

Dhaka	Mina
September 29, 2019	0413052108

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Named Data Networking (NDN) is the named based networking system for next generation network architecture. In-network caching in NDN is a very challenging problem due to computational cost for finding the proper node to store the content in the resource constrained NDN. Moreover, limited resource caches and the distributed locations of cache storage have made the problem complex. NDN content caching is done in network layer which makes packet forwarding and routing more difficult. The important part of NDN is to find the best locations to cache the best content. In this research we analyze some content caching strategies in NDN and put forward a caching strategy based on content popularity, then compare the effect of different caching strategies through simulation experiments. In our research, we propose a caching policy what we call Load Balanced Collaborative Content Caching (LBCC) strategy in NDN based on content popularity. To compute content popularity we use two deciding factors which are content request rate and request pattern. Cache performance is estimated in provision of cache hit ratio. We find that in our method cache hit ratio, response time and interest satisfaction ratio are better.

Chapter 1

Introduction

In this chapter, we provide the introduction of our research field. Here, we briefly describe the name data networking, the architecture of internet and NDN, major components of NDN, network caching in NDN and its role, advantages and challenges of in-network caching, novelty of our contribution.

1.1 Named Data Networking (NDN) Architecture

Named data networking (NDN) is a part of information-centric networking architecture in which data or content is identified by a unique name instead of IP address [kay, VLSY15, AMZ12]. This new architecture is no longer concentrates on where the information is located, but emphasizes on what" or the information (content) is needed. Data are exchanged within the network nodes by specifying their names in *interest* packets and *data* or *content* packets. In NDN, *consumers* send interest packets as a query to request the content they needed. Routers receive the interest packets and search the matched content in their *Content Store (CS)*. If a matched content is discovered, the nodes send the data back to consumers. Otherwise, the nodes forward the interest packages to the next routers using *Forwarding Information Base (FIB)*.

Figure 1.1 shows the hourglass architecture of internet and NDN. NDN uses in-network caching of user requested content. A typical NDN node consists of three components, namely Content Storage (CS), Pending Interest Table (PIT) and Forwarding Information Based (FIB).

- Content Store (CS, a local cache): Content Storage (CS) is one of the important components in the NDN router node. CS work as a storage unit that is established structure in all the nodes. At the point when a node gets a content, based on the standard cache strategies of the node, system will store a copy of this content in its content store (CS). CS is one of the limited resources on NDN routers. Therefore, it should be utilized as efficiently as much as possible in order to improve NDN performance.

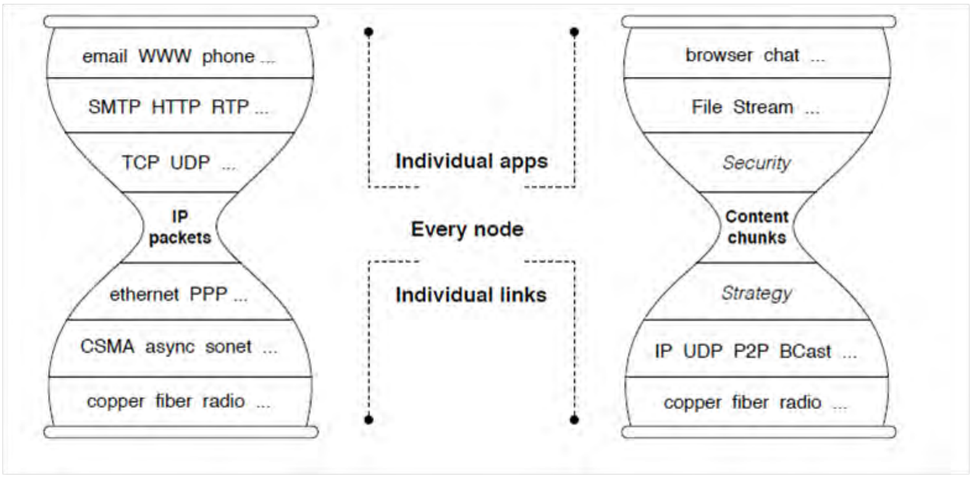


Figure 1.1: Internet and NDN Architecture [kay].

- Pending Interest Table (PIT): It records all requests arrived at the router those not served yet. A NDN router that receives the request from the consumer, will check whether the content is in its CS. If the requested data is found there, the router will immediately send the requested data to the consumer. If the requested data is not found in CS, then the router checks the PIT to match the request of that content previously and not served yet. If in PIT, there is such information then the information will be updated by adding information that other consumer also requested the same data. The information on this PIT makes a reverse path for sending data to the consumer. PIT contains two main processes. At first, it temporarily stores the requested packet before transmitting them to the next node. By using this table, the content can explore the invert ways to achieve the customers who have requested for the data.

- Forwarding Information Base (FIB): FIB handles *Interest* packets forwarding. FIB maintains a table like IP routing table. Based on this table, it transfers the information of data request messages to sources those have the requested contents. If in the PIT there is no same data request from other consumers, then checking is done in Forwarding Information Based (FIB). The interest packet will be forwarded to the data provider node according to the information in FIB. If FIB does not store the content provider's node data, the interest packet will be discarded by the NDN router.

1.2 How NDN Works

Consumers firstly send out *Interest* packets. Routers keep track of the pending interests using *Pending Interest Table (PIT)* to guide *data packets* back to the consumers. Data packets are returned back along the equivalent way in the inverse direction.

For example, five routers R1, R2, R3, R4 and R5 are connected in a network (shown in Figure

1.2). There are two consumers and one data provider. Firstly, consumers request for data by sending an interest packet through the network. Providers receive the interest packet, and send the data packet through the reverse path. When data are back to the reverse path all router are on reverse path cache that data. In the example we find the following paths.

Interest packet path: R1 → R2 → R3 → R4.

Data packet path: R4 → R3 → R2 → R1.

Caching nodes are (R1, R2, R3, R4.) When second consumer sends the interest request for the same data through the router R5 and R2, R2 provide the expected data packet from its cache.

Figure 1.2, Figure 1.3, and Figure 1.4 show the step by step working process of NDN.

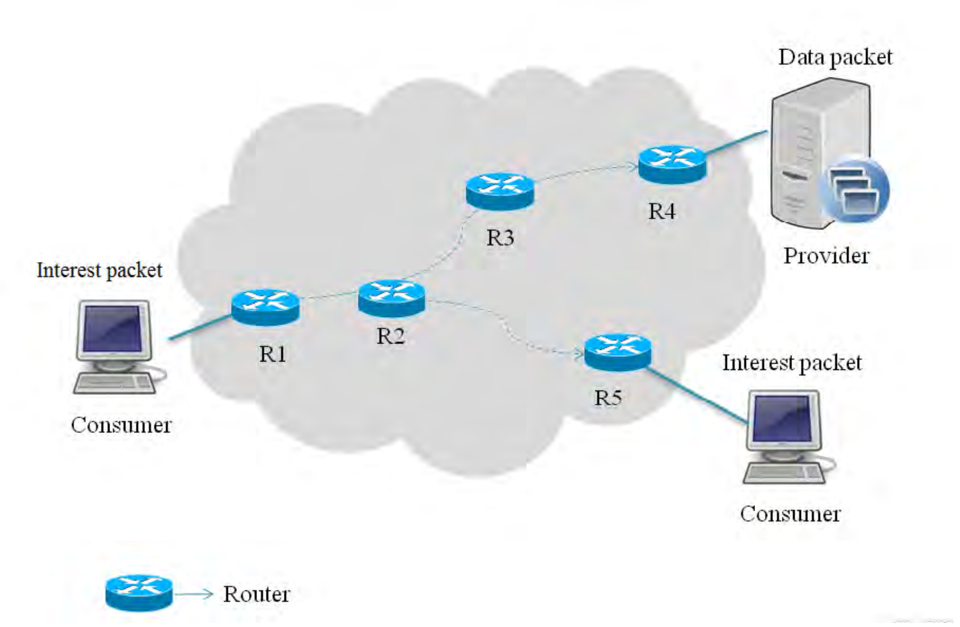


Figure 1.2: Step 1: Working process of NDN.

1.3 Interest Forwarding Process

Figure 1.5 shows the detailed forwarding process of NDN by an activity diagram. At the first of Step 1, consumer sends a interest packet to the intermediate router. In Step 2, router looks into CS table. If matches are found in CS table, than data is provided to the nodes which requested for it. In Step 3, PIT lookup is done. If matches are found in PIT, then interest is discarded. But if there is no match found in PIT, then Step 4, lookup in FIB is done. If matches are not found in FIB, then the interest is discarded. But if matches are found in FIB, then in Step 5, it is added to PIT. In Step 6, interest is forwarded. In Step 7, lookup is done CS table is done for matching. If it is found, then data is forwarded in Step 8. In Step 9, PIT is checked for unsatisfied pending requests, then it is added to CS table in Step 10. Finally, data is forwarded to consumer.

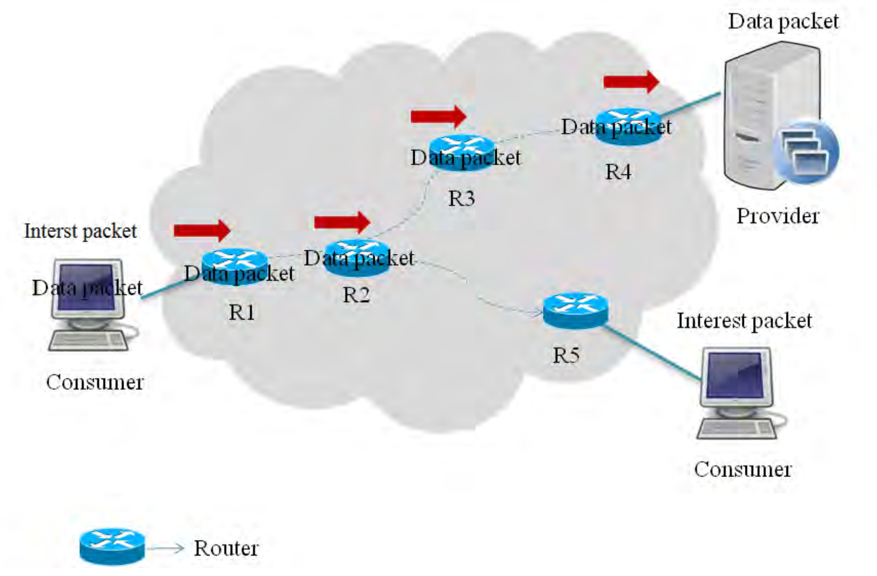


Figure 1.3: Step 2: Working process of NDN.

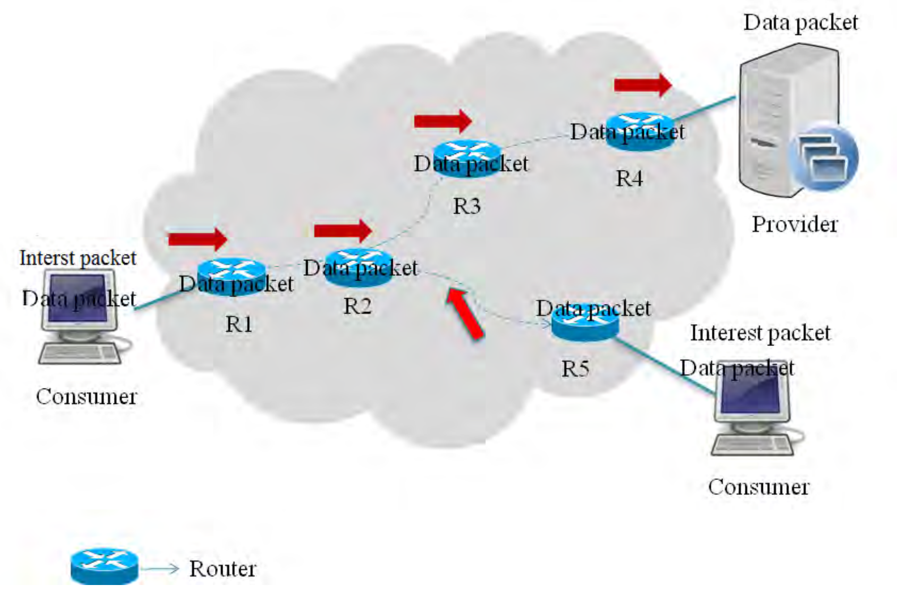


Figure 1.4: Step 3: Working process of NDN.

1.4 Caching in NDN

An important feature of NDN is to manage the in-networking caches with caching strategies.

1.4.1 In-Network Caching

Content caching at intermediate nodes is called *in-network caching*. Request catching and content catching appear in NDN. Cache capacities of routers are relatively small compared with delivered data size. Figure 1.6 shows *in-network caching*.

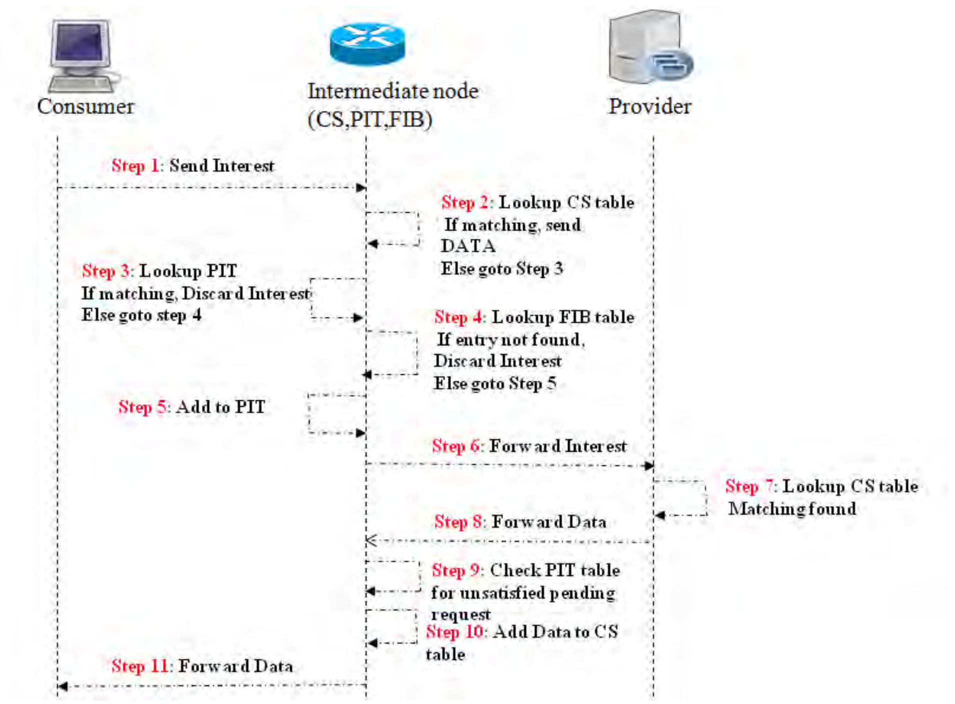

Consumer
Intermediate node (CS,PIT,FIB)
Provider
Step 1: Send Interest
Step 2: Lookup CS table If matching, send DATA Else goto Step 3
Step 3: Lookup PIT If matching, Discard Interest Else goto step 4
Step 4: Lookup FIB table If entry not found, Discard Interest Else goto Step 5
Step 5: Add to PIT
Step 6: Forward Interest
Step 7: Lookup CS table Matching found
Step 8: Forward Data
Step 9: Check PIT table for unsatisfied pending request
Step 10: Add Data to CS table
Step 11: Forward Data

Figure 1.5: Interest Forwarding Process.

Types of In-Network Caching:

- On-Path Caching: In this method, the retrieved contents are cached at the intermediate routers those fall on the symmetrical way back from server to the requester nodes. Figure 1.7 shows the On-path caching technique.
- Off-Path Caching: Here, nodes within a specific domain or all the nodes in the network are utilized to cache contents collaboratively. And requests can be forwarded to the nearest copy which is not on path. Figure 1.8 shows the Off-path caching technique.
- Edge Caching: Here, nodes which are situated on the boundary position of a network, these nodes are enabled to cache the content. Figure 1.9 shows the edge caching technique.

1.4.2 Benefits of In-Network Caching

In-network caching is popular in NDN because of some unique benefits described as follows. Firstly, intermediate routers can share the concern of providing contents which will lighten the load of the original content servers.
Secondly, peer-to-peer traffic involves minimum paths and this may reduce the risk of congestion. Finally, the response time and the transmission overhead for conveying contents are minimized because the required contents can be found from the nearest cache instead.

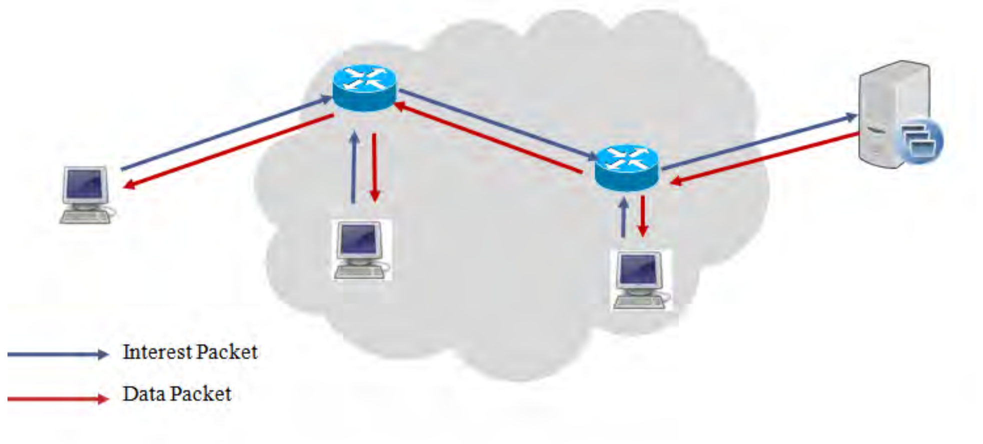


Figure 1.6: In-network caching.

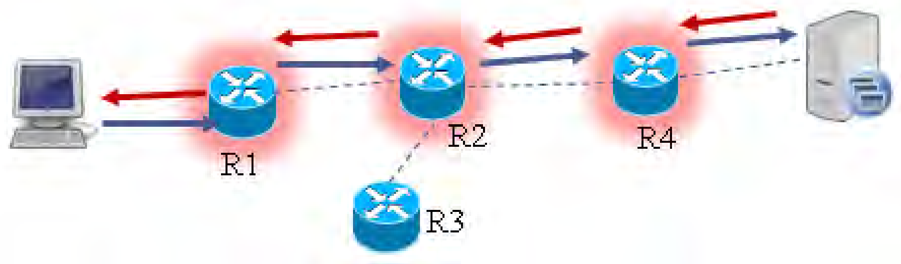


Figure 1.7: On-path cache.

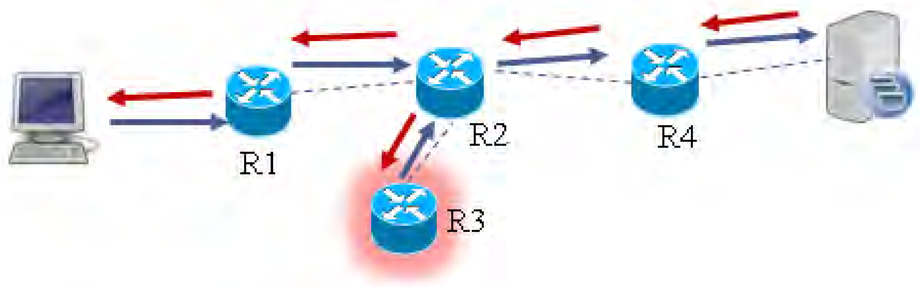


Figure 1.8: Off-path cache.

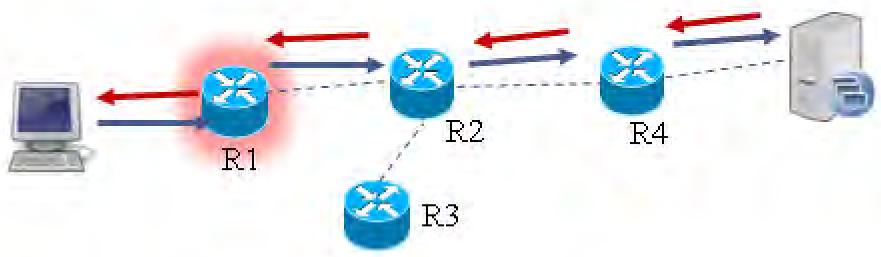


Figure 1.9: Edge-path cache.

We can summarize the advantages as follows.

- It reduces the unnecessary fetching of content from the original content server.
- It improves user response time.
- It reduces data access latency and network load since content are stored intermediate nodes.
- It improves network resource utilization.
- It is cost effective data retrieval process.

1.5 Challenges in In-Network Caching

In this section, we provide the some challenges for in-network caching method.

- *Cache Placement or Allocation*
  It is a challenge where to cache the content (i.e. content stores) Selected nodes may be Edge nodes / core nodes / central nodes / strategically selected nodes
- Cache Size Dimension : What should be the size of caches? To select the homogeneous or heterogeneous caches is very challenging task. In case of heterogeneous where to raise cache size relatively
- Content Placement : Where to store a retrieved data within a network? Cache the retrieved content for better performance is very challenging due to identify the Centralized or decentralized manner
- Content Packet Selection: What to cache from the huge flow of content packet? To identify profitable contents from the huge volume of contents. Content request patterns repeatedly change from time to time (thus content-object popularity also changes)
- How long the content will be stored? - There are issued how long the content will be cached. (short time lived vs. long time lived content-objects)
- There are some challenges in handling dynamic nature of real-world content-object requests.

1.6 Motivation

An important feature of NDN is to manage the in-networking caches according to caching strategies. Several cache management techniques appear in NDN. For example, MAGIC or maximum-gain in-network caching [RQ+14], hop-based probabilistic caching (HPC) [WXF13], most popular cache (MPC) [ZWWQ17], and ProbCache [PCP12].
The default in-network caching strategy is Leave Copy Everywhere (LCE) [kay] caches all the contents in all en-route routers. It cause two problems as follows: The adjacent routers cache the same contents that have caused a large amount of redundant data in the network and reduce the diversity of data. The routers on the path of data replying are treated equally to cache same contents and the importance of each routers location is not considered. It greatly wastes the storage space of caches on the routers, and as a result, the load in not balanced.
The authors of [PCP12] proposed ProbCache- the strategy takes account of distance from content source to users, and caches content at the users nearest router with larger possibility thus reducing transmission delay when other users request the same data.
Age based caching strategy [MXW12] reduces the network delay and traffic. but it works on only for read-only objects. Age-based caching still cannot avoid the redundant caches.
Centrality, is a graph theory-based concept is used to social network to find important nodes in a graph. A high *centralilty score* is given. The sizing the content store is based upon centrality. In this centrality based caching strategies, we find several centrality deciding parameters. For example betweenness centrality, closeness centrality, stress, graph, and degree etc. to heterogeneously distribute content at routers instead of homogeneous distribution. Simple degree centrality based allocation is proposed that it is sufficient to distribute content store. Similarly, [YGS16], if content is stored in between centrality nodes, then it offers a higher cache hit ratio. But it does not mention any content popularity.
CL4M [CHPP13] perceives the state of topology and chooses the router with the largest betweenness centrality as the cache router. The network topology is considered in Prob-Cache and CL4M, while the popular contents still cannot be acquired quickly.
Literature's [YLLL17, WSG+14] take both content popularity and node level into account. CPRL Strategy [YLLL17] proposes a popularity based caching strategy which is based on content popularity and matched node level. *Popularity* of content has been calculated in advance according to *Zipf* law, where content popularity is static, which does not take the dynamic changes of the content requests into consideration in the network. It cache only one content in a router and it does not define the caching position for same or corresponding popular content. [WSG+14] designs a dynamic, self-adaptive method to calculate the content popularity and proposes a novel caching scheme (CRCache) that utilizes a cross-layer design to cache content in a subset of routers that depends on the correlation of content popularity and the network topology. However, [YLLL17, WSG+14] needs a lot of extra information or topology of the entire network when calculating the router level, which increases the storage and computing

overhead of routers.

We enlist major findings based on the major drawbacks:

- In popularity based caching strategy, we face problem when the popularity is same for two or corresponding contents. It becomes challenging what to cache.
- Cache hit ratio is low in collaborative caching framework because of its architecture.
- Time factor is missing on-path based caching.

1.7 Our Contributions

In order to solve this challenging problems, we propose an in-network caching strategy named load balanced collaborative content caching strategy (LBCC) based on content popularity to improve the overall performance of NDN network. The strategy makes full use of the popularity of the content, cache node selection based on hop count and content popularity and then selectively caches the contents according to their popularity on the selected routers. It improves the utilization of cache resources and reduces the frequency of contents replacement. In addition, on the path of data replying, only one router will be selected to cache the same type of contents. It avoids the same data caching on the adjacent nodes, reduces the amount of redundant data in the network, increases the diversity of data, and improves the cache hit rate.
We summarized the our contributions as follows:

- We propose a popularity-based collaborative content caching scheme for NDN, which combines popularity based on-path caching decision and time duration constant.
- The popularity estimation and comparison process are optimized as the requirement of quick process for in-network nodes. To inuence the popularity estimation there is also a detailed analysis of the overhead and statistics of cases.
- We propose a popularity based content caching strategy that can significantly boost cache performance. We use *Request Trend (RT)* to calculate position of the content and its relative measure.
- We use time factor *k* to cache most popular content for a time.
- We propose a cache node selection algorithm for caching popular contents near the consumers.
- Using NDN standard simulator ndnSIM, we measure the performance of our proposed model. For this, we use different standard performance measures.

1.8 Organization of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 presents the related work and their drawback. The detailed proposed strategy is illustrated in Chapter 3, and the simulation results and performance measures are shown in Chapter 4. Finally, Chapter 5 provides the conclusion.

Chapter 2

Related Work

Caching in NDN plays vital role because it ensures the availability of data to the network users based on the stored appropriate data in the storage. Therefore, data or content must be available whenever the user requests for it. Researchers are very interested to find a suitable caching strategy.

Caching is the most productive methodology that can be utilized for accessing information oriented services and technologies to reduce the data traffic and storage overhead. Named data networking uses caching technique in several levels based on the different caching strategies. In this chapter, we discuss several caching strategies, caching types, their advantages and disadvantages.

2.1 Probability Based Caching Strategy

Probability based caching strategy is mainly the caching node location selection strategy based on some probability computation. Caching content is done at every node towards the conveying path with a specific probability.

Probabilistic Cache (ProbCache) offers higher probability to store the contents when a node is closer to the destination nodes [PCP12].

Hop-based Probabilistic Caching (*HPC*) uses the number of the hops between the content providers or intermediate nodes and the router to regulate the caching probability [YGS16, WXF13]. Figure 2.1 shows the probability based content caching where user requests for a content to server via path R1, R2, R4. Content caching on delivery path on router R1 and R4 are done based on certain probability.

Drawbacks of this methods are described as follows.

- It omits the physical location of routers when making caching decision.

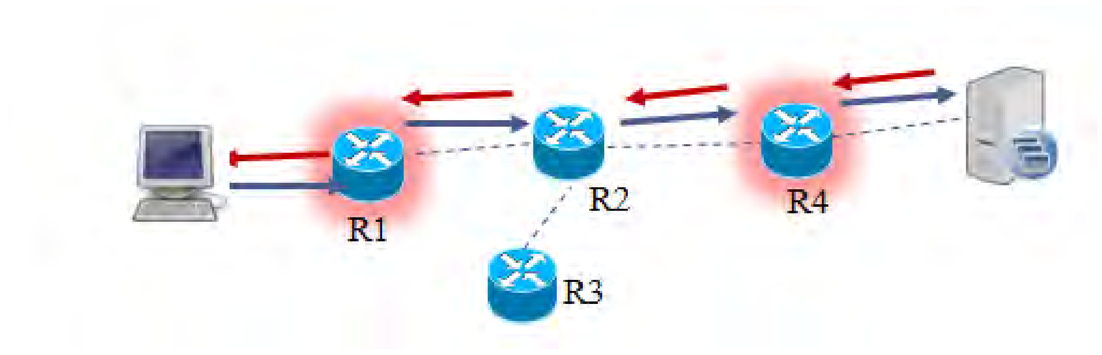


Figure 2.1: Probability based content caching.

- It always allow more cached contents closer to the destinations an it causes overload to those routers [YGS16, PCP12, WXF13].

2.2 Popularity Based Caching Strategy

Caching the content is done based on the popularity of the content. We can mention two types of popularity-based caching strategies. These are *Content Popularity* and *Users Popularity* content caching discussed by authors [YLLL17, CHPP13, LWL+12, WSG+14, RQ+14, YL18].

Figure 2.2 shows the popularity based content caching framework where a popularity table is added on server side, and popularity is measured based on the user requests. Here content c3 is most popular, content c2 is more popular and content c1 is less popular. Based on popularity table router R1 caches c3 content and R4 caches c2 content.

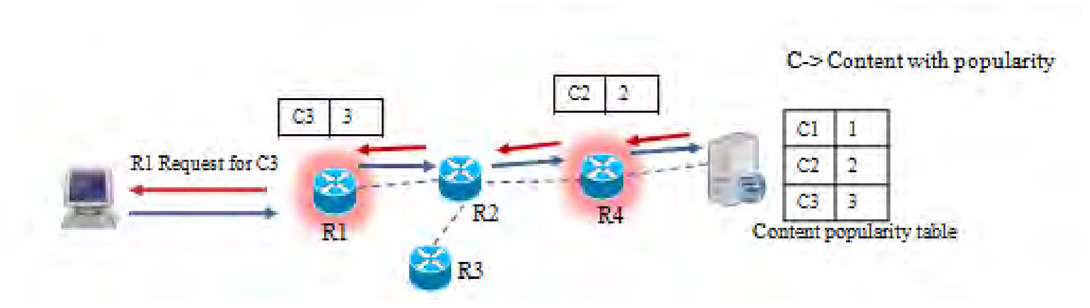


Figure 2.2: Popularity based content caching.

Major drawbacks of the system are described as follows.

- It causes too much occupancy in edge routers for caching which leads to high overhead there.
- It may use only elected nodes in network backbone more than network edge router where quick response can be offered.

- The computation cost is very high due to calculation the popularity of each content at each node.

2.3 Age Based Caching Strategy

In age-based caching [MXW12], the contents are marked with age in the network. The principle of the scheme is as follows.

- Each piece of content has an fixed time which is called *age*. The lifetime of a replica in a router is decided by its age.
- Content copy obtains its age when it is added into the cache, and is removed from the cache when the age expires.
- Routers implicitly collaborate by modifying the age.

Age is used to decide the caching which agrees the following rules:

- The closer a copy is to the network edge, the longer age it has to stay.
- The more popular a copied content is, the higher age it has.

Figure 2.3 shows how the age based scheme works. In this figure, one client and one server are connected by two routers node (R1 and R2) [MXW12]. The server provides mainly two types of contents, most popular content and less popular content. The consumer gradually requests the two contents according to their popularity. For the intentions of simplicity, we assume both that routers have the storage capacity of copy one content. We start at the initial phase that two routers have no content cached Figure 2.3 (a). At a certain time, the popular content is requested by the consumer and both routers have the content cached, which leads the system to Figure 2.3 (b). According to this scheme, for the same content, R1 assigns it a longer age than R2. After some time, the popular content in R2 expires but R1 still cached the content (shown in Figure 2.3 (c)). If at this time, the less popular content is required, then R2 will obviously cache the copy of the requested content. Now, the system migrates to Figure 2.3(d). This status is optimal, as aggregated network delay and publisher load are minimized. Because we provide the popular content with longer time than the less popular one, the popular content will have less opportunity to be replaced, which tend to maintain the system in this state.
This policy adaptively attempts content objects to the network edge by using dynamically changing caching time that controls the lifetime of the content copies in the router. The time of the content copy is determined along the routing way and there is no signaling information between nodes or extensive computational overhead. In fact, it spreads popular contents to the

network edge and simultaneously eliminates unnecessary copies of content in the middle of the network.

Drawbacks of the strategy are described as follows.

- This is applicable only for read-only objects.
- Age-based caching still cannot avoid the redundant caches.

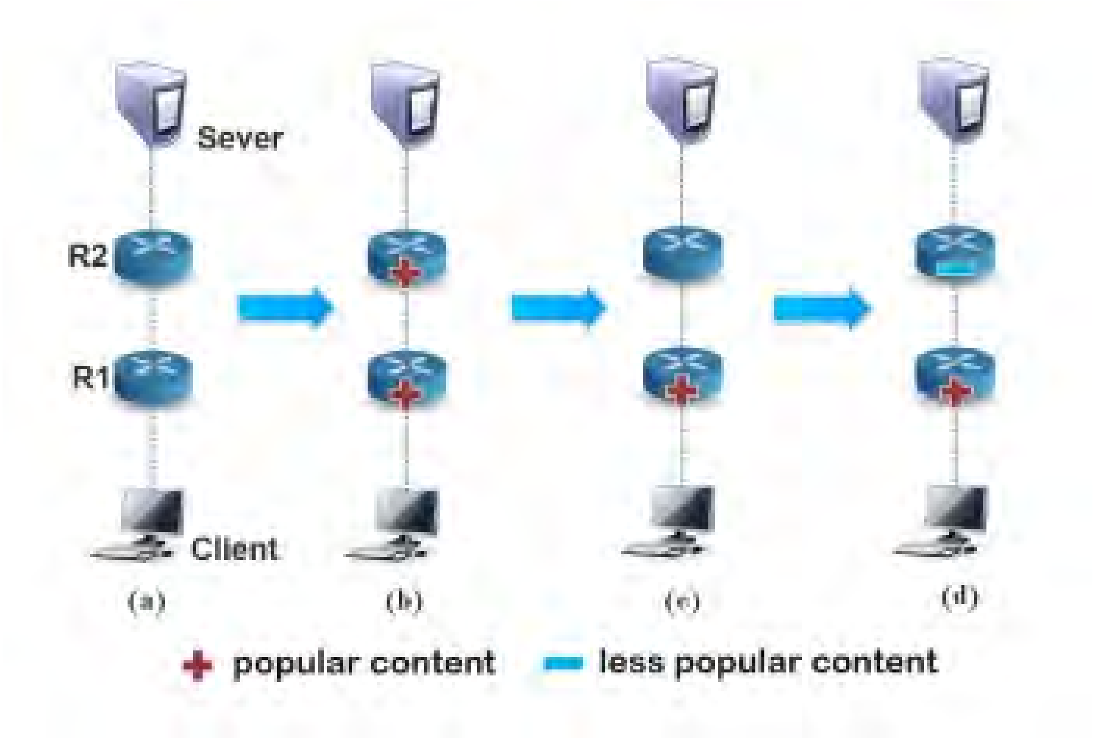


Figure 2.3: Age Based Content Caching [MXW12].

2.4 Collaborative Caching Strategy

Instead of individual decision making for caching, collaborative caching strategy attempts to introduce collaboration among routers to make caching decision for lower redundancy in content caching. This method usually involves in a certain message exchanges among devices to jointly make caching decision.

Age-based cooperative caching [MXW12] extends popular contents to the network nearest node by progressively changing the caching time of contents in an implicitly cooperation manner. In [WZB13], an intra-AS cache cooperation scheme is proposed to offer neighbor routers to discard redundancy after storing content and combine in assisting each others requests. Intra-Domain cooperative caching (IDCC) scheme in [JXY14] combines probabilistic based caching and gradually caching. In IDCC, cached content advertisements are diffused in the intra-domain to

reduce cache redundancy for enhanced cache utilization.
In [DHL+12], authors explore the capacity populating problem and propose a collaborative hierarchical caching technique accordingly.
Popularity-based neighborhood collaborative caching scheme, combines popularity based on-path caching decision and assist caching by using the bloom lter [ZWWQ17]. For the requirement of quick process for in network nodes the popularity estimation and comparison process are optimized. In the evaluations to examine the performance of algorithms [ZWWQ17] real world internet topologies are used. This scheme is compared to not only efficient existing on-path and o-path algorithms but also the ideal situation that requests can be forwarded to the closest copy in the network with no extra information and query delay [ZWWQ17].

A simple example of caching decision is shown in Figure 2.4. Each node can store one content and the popularity rate is A >B >C >D >E in this example. And the current cache status is also shown in Figure 2.4. The client sends a request for content B into the network. There is no change to the request as the popularity of B is lower than A at R3. Then the request is forwarded to R2. Content B is not appears in R2 or one-hop neighborhood nodes of R2. As the popularity of B is higher than C which is stored at R2, R2 changes the compare tag of the request to false and adds its name to the request. After completing the above process the request is forwarded to R1. and we see that, there is no cache hit at R1. In router R1 there is no cache decision as the compare tag has been changed to false. The request is nally forwarded to the server to get the content B. And content B will be stored at R2 when it comes back to the client with the name of R2 stored in the content B.
The main principle of this scheme is given bellow:

- Individual decision is made for caching [MXW12]
- Collaboration among routers are made for caching decision for minimizing redundancy in content caching [WZB13].
- It involves a fixed message exchanges among devices to combine making caching decision [MXW12].
- It dynamically configure contentś age among NDN routers [MXW12, WZB13, JXY14, DHL+12].
- The copy acquires its age when it is added into the cache.
- The copy is removed from the cache when the age time-out.
- Routers implicitly collaborate by modifying the caching time i.e age [MXW12, WZB13, JXY14, DHL+12].

Drawbacks are as follows.

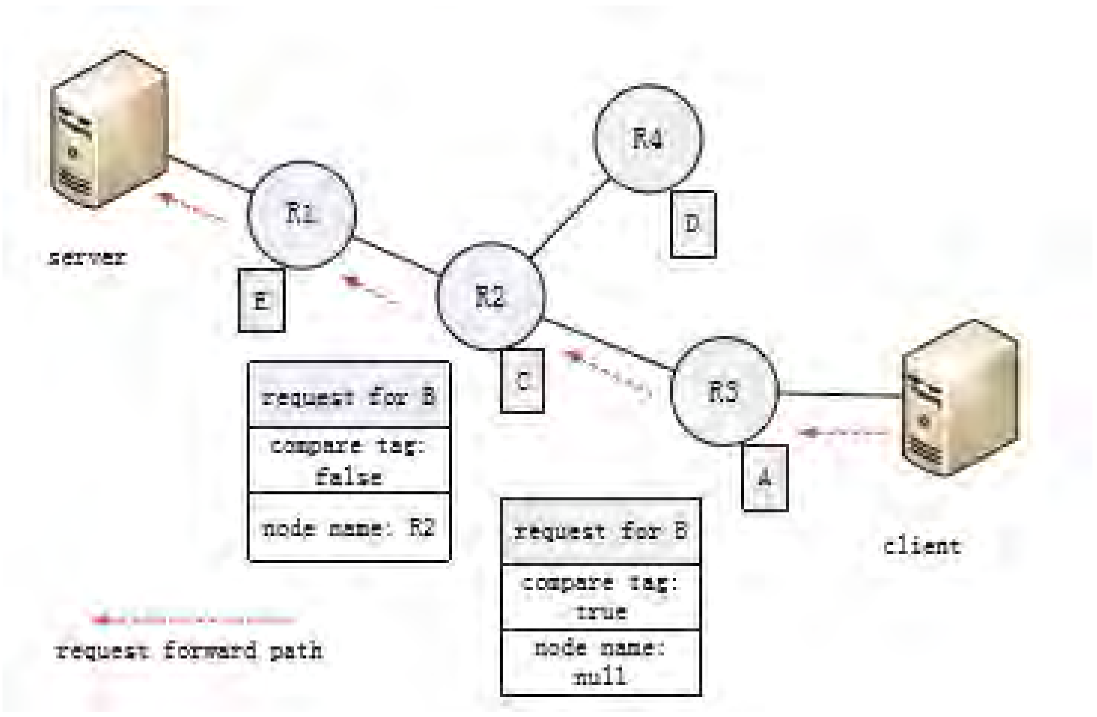


Figure 2.4: Popularity Based Collaborative Caching [DHL+12].

- Collaborative caching approach needs message exchanges which introduces additional overheads [MXW12, WZB13, JXY14, DHL+12, ZWWQ17].

2.5 Centrality Based In-Network Caching Strategy

Universal caching strategy is unnecessarily costly and sub-optimal. High content replacement frequency may result in content being replaced before getting a hit. *Cache less for more* scheme is a popular centrality based method. This scheme enhance the overall content delivery performance by caching only specific subset of nodes en-route the delivery path [CHPP13]. This scheme is based on the concept of betweenness centrality [CHPP13] which determines the number of times a specific router which stay on the content supplying path between all pairs of nodes in a network topology [CHPP13, LWL+12]. The basic idea is that if a node stays along a high number of content delivery paths, then it is more possibility to get a cache hit. By caching only at those more *important* nodes, these reduce the cache replacement rate while still caching content where a cache hit is most probable to happen[8]. Figure 2.5 shows that at time t=0, all cache node stores are empty and client A requests a content at s1 and at t=1, client B requests a content from S1. If frequent caching happens at v1,v2,v3 and v4, v3 becomes necessary for the request from consumer B. v3 being the important node in this case. This can be verified by using the betweenness centrality, whereby v3 has the highest centrality value because it lies in delivery paths of the network.
Drawbacks of this method are described as follows.

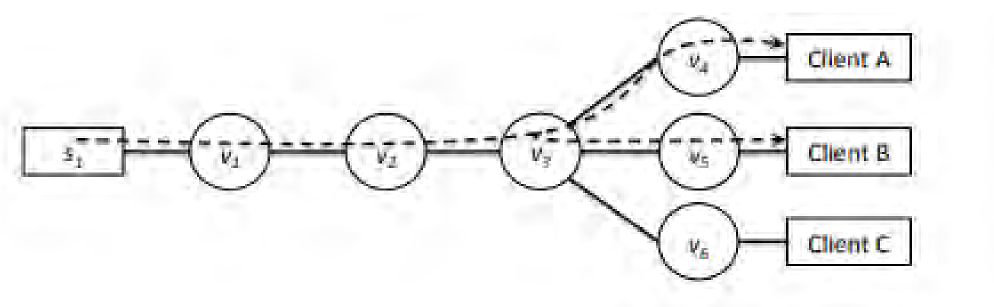


Figure 2.5: Cache less for more [CHPP13].

- It caches content only at the top centrality routers, which will increase the load on that router.
- It causes high frequency of cache replacement.

2.6 MAx-Gain In-Network Caching Strategy

MAx-Gain In-network Caching (MAGIC) [RQ$^+$14] introduces benefits of number of hop reduction and cache replacement penalty for caching content. Contents are identified for caching using the utility to maximal of the cache gain [RQ$^+$14, MXW12]. To do this, authors compute cache placement gain (the gain if caching a new content at the node) and cache replacement penalty (the loss if evicting a cached content from the node) by jointly evaluating the content popularity and hop reduction which deliberates the saved bandwidth consumption. This method defines the effect of local cache gain for a node which combines the effective cache placement gain and cache replacement penalty to minimize the number of caching operations. Finally, the authors designed a distributed caching technique MAGIC to cooperate in-network caches along a path.

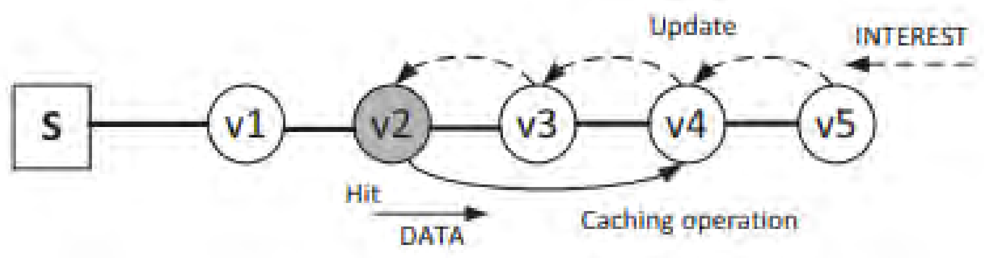


Figure 2.6: MAGIC Operation [RQ$^+$14].

Figure 2.6 shows the basic operations of MAGIC which operates per request. To determine the router with the maximal *local cache gain* for content, the special message to carry the maximal local cache gain along the path in the newly defined *maximum gain* field. To request content, an

INTEREST packet is sent and *MaxGain* value is set to 0 in the header. When the INTEREST packet is received, each router computes *LocalGain*, and compares it with the value recorded in the MaxGain portion. If the local cache gain of router is larger than the recorded MaxGain value, router will update the MaxGain value in the INTEREST packet. When the INTEREST message reaches the original server, the MaxGain field in the INTEREST message is the maximum local cache gain along the delivery path.

The major drawbacks are as follows.

- MAGIC is very costly because it needs to calculate each content at each node.

As discussed above, it is a very challenging problem to find the proper node to cache the proper content. In order to make more popular contents near to the user, replica is needed which introduces redundancy. If replication is least, computation and search time is increased. Therefore, we propose a load balanced content caching strategy based on content popularity and using some collaboration information about the content. The content will be located hierarchically by the popularity and load is balanced by caching unique content in all routers.

Chapter 3

Problem Domain

In this chapter, we provide the formulation of the problem and discuss the details of our solution approach.

3.1 Preliminaries

An important feature of NDN is to manage in-network caching with different caching strategies. However, finding a good in-network caching strategy is a very challenging problem. We now present some preliminaries which are required to formulate the problem. Our proposed framework for content caching can significantly boost cache performance. It comprises of two main components which we call *Request Rate* and *Request Trend*.

- *Request Rate (RR)*: It is the total number of individual requests of a content divided by overall request counts. In our strategy, we find he initial popularity of that content from the Request Rate.
- *Request trends(RT)* means the content request patterns in a period of time. For example, a *content trend* refers to the most used keywords denoting the content on the network during a given period of time. A *trend* is a subject that appears on one or more for a limited duration of time. There is no limit that how long a content stays as popular.

We can explain with an example as follows. We find the request on Router 1 for content *A* and content *B* on the *Content Request Table* shown in Figure 3.1.

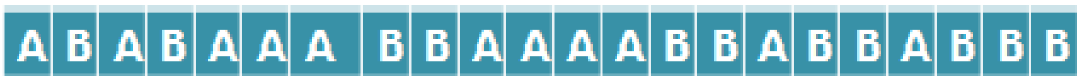


Figure 3.1: Requested content.

Now, we show how to use these two parameters. We compute these two values.
Request Rate(RR) Computation: We compute RR from number of user requests. It is calculated by mean value of user requests. For Example, in Router 1 for content A and content B be on the content request table shown in Figure 3.1, the request rate is calculated as follows. Request Rate (RR) for content *A* is : 11/22 = 0.5(Initial popularity of content *A*)

Request Rate (RR) for content *B* is : 11/22 = 0.5(Initial popularity of content *B*)

Here, Popularity ranking of content A is 1 and content B is also 1, that means it shows same popularity rank for content A and B. It is difficult to find the caching position for same popularity ranking, to solve this problem we use another factor named Request Trend.

Request Trend (RT) Computation: Request Trends are related to the appearance of the same content request. Let us find the all continuous content blocks. Therefore, *Request Trend = Individual Trend / Total Trend*

Algorithm 1 show the details process of computing *Request Trend*, where k is a constant for time duration for the current popular content. Based on k, we increase the caching time for accommodating more popular contents.

Algorithm 1 Request Trend Calculation

```
Initialization:
Trend = 1;
TrendsValue = 1;
**if** Current Request == Last Request **then**
    Trend = Trend + 1;
**else**
    **if** Trend >1 **then**
        TrendsValue = Trend * k; [k=Constant]
        Save to table
    **end if**
**end if**
```

According to Algorithm 1, individual trend is computed when we find current request and the latest request is the same, and then trend value increases for the same content. If the content is not the same and trend value is greater than 1 than we multiply the trend value to content value k and then trend value is saved to trend table.
We can take a look on our content request table in 3.1. In this figure, the trend value for A is computed according to the following procedure. Firstly, we match the current request *A* with the last request, since *A* is the first content than last value is null, so nothing to add here. Now, current request is *B* and *A* is not equal to *B* then we go to *A* and we find *B* is not equal to *A*. When matching is found, we increase the trend value 1. But when *B* is found and trend is greater than 1 then multiply the time duration constant K (where K=1) with trend value and we save the value 3 for *A* on the table. In this way, we find the individual trend value for all contents and Table 3.1 shows the trend calculation results.

For Example, content *A* and content *B* on the content request table are shown in 3.1. Here, we find that A is found 3 times continuously, B is found 2 times, then again A is found 4 times and so on. All results are shown on Table called *Trend Value Table*.

Table 3.1 shows the trend calculation results.

Table 3.1: Trend Value

Content	**Trend**
A	3
B	2
A	4
B	2
B	2
B	3

After finding the trend value for all contents, we calculate the content trend by using following formula:

Content Trend = *Individual Trend* / *Total Trend*

For content *A*,

Trend = (3+4) / (7 + 9) = 7 / 16 = 0.44

For content *B*,

Trend = (2 +2+2+3) / (7 + 9) = 9 / 16 = 0.56

3.2 Update Popularity

In this section, we show the how to update popularity.

Updated Popularity = (Initial Popularity + *Request Trend*) /2

For content *A*, popularity = (0.5 + 0.44) / 2= 0.47 (Popularity ranking 2)

For content *B*, popularity = (0.5 + 0.56) / 2= 0.53 (Popularity ranking 1)

We find that update popularity of content A is 0.47 and B is 0.56, if we rank the popularity then we find that popularity rank is 1 for content A and rank 2 for content B based on the highest update popularity. This popularity ranking solve the same content popularity caching problem and provide unique content cache on each router.

Content *A* and content *B* are cached on Router R2 and Router R1 correspondingly based on the proposed method shown in Figure 3.2.

We now, provide the next step in Algorithm 2.

After finding the updated popularity of the content, we add the update popularity on the *edge router node* and set the hop value zero for this router. The other routers who have received

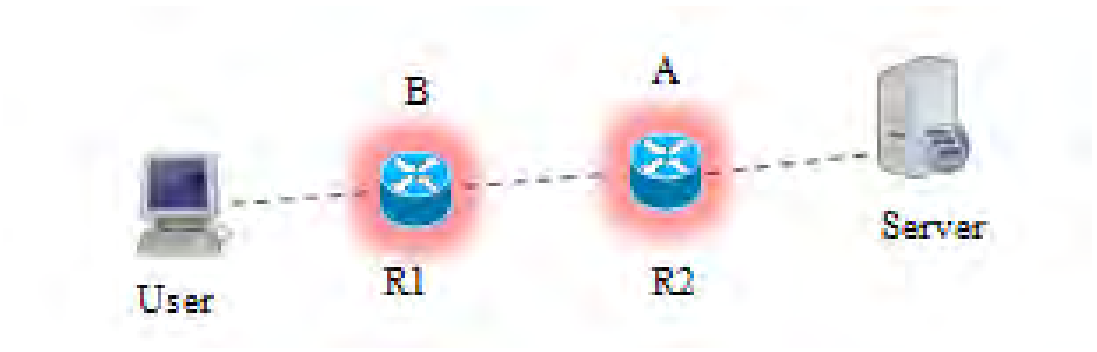


Figure 3.2: Update popularity.

Algorithm 2 Interest Request Method

```
for Each (received interest packet) do
    Calculate content popularity
    if (Data matching is found in CS) then
        It sends (data)
    else
        Hop count is increased.
        if (Find matching content name in PIT) then
            Add interface of previous router to PIT
        else
            if (Find match content name in FIB) then
                Forward interest request according to FIB
            else
                Drop the interest request
            end if
        end if
    end if
end for
```

the interest packet will just forward the interest packet after adding 1 to route hops count. The operation of the Interest request phase and the related pseudo-code is shown in algorithm 2.

3.3 Node Selection

Basically, a user requests a content, edge router calculates the content popularity. After finding the updated content popularity, the routers on the forwarding path count the number of hops from the user. From content popularity and hop count, router calculates the caching position. The routers which are closer to the users will be assigned the highest priority, and the router that is farther from the user will get less priority. The popularity level of the routers is available only during a certain request and reply period on the path. Since, we work only on-path routing, other routers in the network will not be assigned router level when they are not on the route.

Algorithm 3 shows the details process of node selection method for content caching.

Algorithm 3 Find Caching Router Position

```
while Get Popularity position do
    if Popularity Rank is less than or equal to hop then
        return Popularity Rank - 1
    else
        return -1
    end if
end while
```

We can explain with an example.

Hop count: R1 = 0, R2 = 1, R3 = 2, R4 = 3
and from Table 3.4 we find the popularity rank of the contents.

For example, we compute the cache position of content *A*.
Scenario 1:
Content A is found on R2; then we check the condition, (popularity rank <= hop) ; (2 <= 1). Condition false, content A is already cached on R2 and there is no need to cache it again.
Scenario 2:
Content A is found on R4; then we check the condition, popularity rank <= hop ; 2 <= 3; condition true; so, cache node position for content A is (2-1) = 1 Therefore, the caching node is R2.

Accordingly;
Cache node Position for content *B* is (1-1)= 0
Cache node Position for content *C* is (3-1)= 2
Cache node Position for content *D* is (4-1)= 3
We now explain. Algorithm 4 shows the details data packet forwarding process.

Algorithm 4 Data Packet Reply Algorithm

```
for Each (received data packet) do
    if (Content match found on PIT) then
        if (Router position is equal to caching router position) then
            Router position found
        else
            Forward data packet according to the PIT
        end if
    else
        Drop the data packet
    end if
end for
```

The caching router positions are shown in Figure 3.3

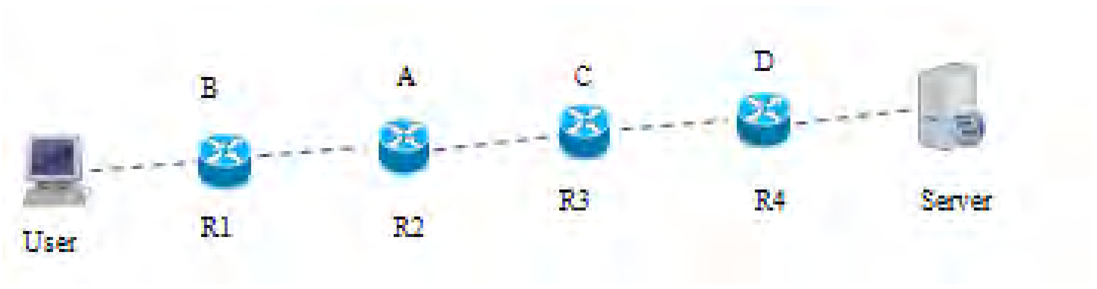


Figure 3.3: Cache Router.

3.4 Our Solution Framework

Our proposed caching strategy which progressively caches the content with the high popularity in the proximity router nodes closer to the users in order to improve the *cache hit* rate and to reduce the average request time. This strategy aims to cache the content according to its popularity level on the corresponding node. Request counts of contents with high popularity will continue to increase, and the popular content will gradually move to the nodes nearest to the users.

3.4.1 Data Tables of Nodes

We use four table structures: CS, FIB, Pending Interest Table (PIT) and a new table named *Consumer Request Table* (CRT) which is maintained by the first level router [YLLL17].
Consumer Request Table (*CRT*): In the first hop, the router which is next to the user, the content Name and User Request Count is stored into CRT for calculating content popularity and cache router node.

We now provide the steps as follows.

- Step 1: The consumers send interest packet to seek the content they needed.
- Step 2: After receiving the interest packet, the first router looks for the number of requests in the CRT table.
  Then it calculates the content popularity by calculating the request rate and request trend according to the Section 3.1 and Section 3.2.
  Then, cache router node according to Section 3.3.
  Then, it adds the content popularity and cache router node in interest packet.
- Step 3: The other routers which receive the interest packet will forward the interest packet after adding 1 to route hops count.
- Step 4: The operation of the interest request phase and the related pseudo-code is shown in Algorithm 1.

- Step 5: When providers which own the content that consumers need receive the interest packet, these will calculate the cache location according to Algorithm 3.
- Step 6: The routers receive the data packet and checked popularity value to cache the data packet or not. The operation of the data reply and the related pseudo-code are shown in Algorithm 4.

For example, we request 200 contents and content flow appears on 3.4

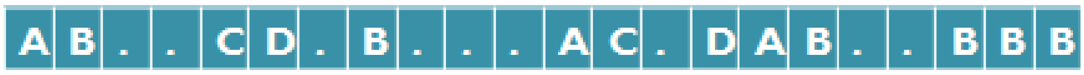


Figure 3.4: User request for contents.

Request Rate(RR) Calculation:
Number of User Requests for A = 40
Number of User Requests for B = 90
Number of User Requests for C = 45
Number of User Requests for D = 25
Total number of requests =200
Therefore, Request Rate = Individual Request / Total Request
For content A, Initial Popularity be = 40/200 = 0.2
For content B, Initial Popularity be = 90/200 = 0.45
For content C, Initial Popularity be = 45/200 = 0.23
For content D, Initial Popularity be = 25/200 = 0.13
Request Trend(RT) Computation
Content Trend = Individual Trend / Total Trend
For content A, Trend = (23+14) / 95) = 37 / 95 = 0.39
For content B, Trend (10+7+2) / 95= 19 / 95 = 0.2
For content C, Trend = (6+13) / 95 = 19 / 95 = 0.2
For content D, Trend (8+12) / 95 = 20 / 95 = 0.21
Updating Popularity
Updated Popularity = (Popularity + Trend) /2
Table 3.4 shows the updated popularity of a content.
Figure 3.6 shows the expected content caching approach.

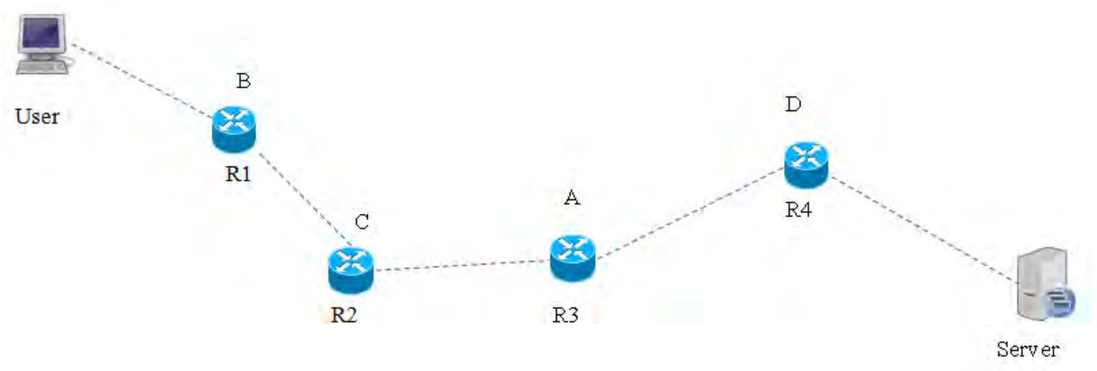


Figure 3.5: Initial content caching.

Table 3.2: Updated Trend Value

Content	Trend
A	23
B	10
C	6
B	7
D	8
B	2
A	14
C	13
D	12

Table 3.3: Initial Content popularity.

Content	Popularity
A	3
B	1
C	2
D	4

Table 3.4: Updated Content popularity.

Content	Updated Popularity	Popularity Rank
A	(0.2+0.39)/2=0.299	2
B	(0.45+0.2)/2=0.325	1
C	(0.23+0.2)/2=0.2	3
D	(0.13+0.21)=0.17	4

Table 3.5: Cache Router Position.

Content	Router Position	Router
A	(2-1)=1	R2
B	(1-1)=0	R1
C	(3-1)=2	R3
D	(4-1)=3	R4

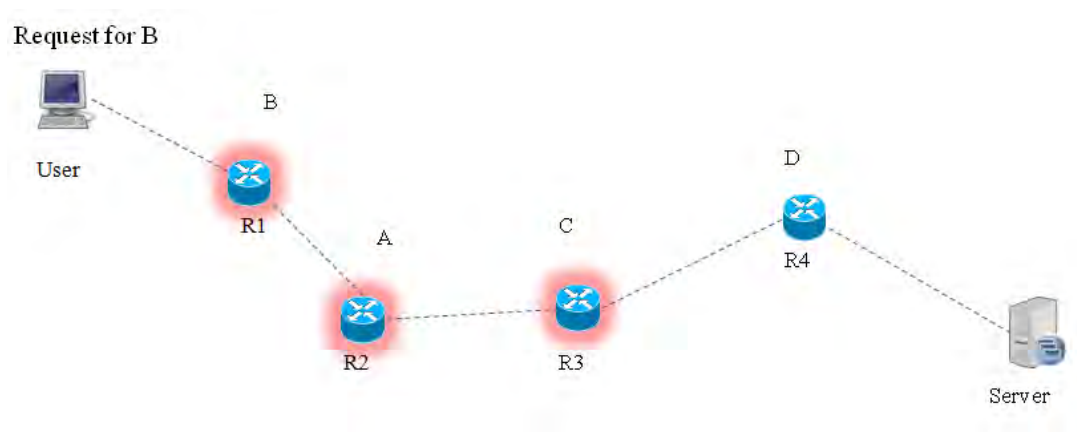


Figure 3.6: Content Caching Based on Popularity Ranking.

After we find the content popularity, we calculate the router position and cache the content accordingly. Table 3.5 shows the router position and caching node.
For example, since B is the last request, we find the cache position of B.
Hop Number:
R1 $\rightarrow$ 0, R2 $\rightarrow$ 1, R3 $\rightarrow$ 2; R4 $\rightarrow$ 3
For example, we find content B on R2. Therefore, we check the condition popularity rank $\leq$ hop; $1 \leq 1$
Condition true.
Therefore, cache node position for content B is (1-1) = 0
Therefore, the caching router node is R1
Accordingly, we find the router position of content A, C, and D.

In this chapter we shows our details proposed method and clarified example of LBCC method.

Chapter 4

Results and Analysis

In this chapter, we provide the results received from simulation study. We use ndnSIM simulator to evaluate the performance of our proposed method *Load Balanced Collaborative Caching* (*LBCC*) strategy.

4.1 Experimental Setup

This section represents the setup in which we have implemented our algorithm and run simulations. We compare our method to some of the existing caching strategies *Content Popularity and Router Level (CPRL)* caching method and widely used *Least Recently Used (LRU)* caching algorithm.
ndnSIM is based on NS-3 simulator that supports NDN communication model. *ndnSIM* is specially extensible to internal structure for NDN implementation.

NDN simulatorś architecture is shown in Figure 4.1 [MAZ17], ndnSIM has been implemented as a network-layer protocol model in NS-3, which can run on top of any available link-layer protocol model. The simulator is implemented in a modular fashion, separate C++ classes are used to model behavior of each network-layer entity in NDN like Pending Interest Table (PIT), Forwarding Information Base (FIB), Content Store (CS), network and application interfaces, Interest forwarding strategies, etc [MAIZ15, SNRJE18].

Next section describes different parameters used in simulations. Our algorithm can be applied to any topology. Here, we use default tree topology of ndnSIM (shown in Figure 4.2.)

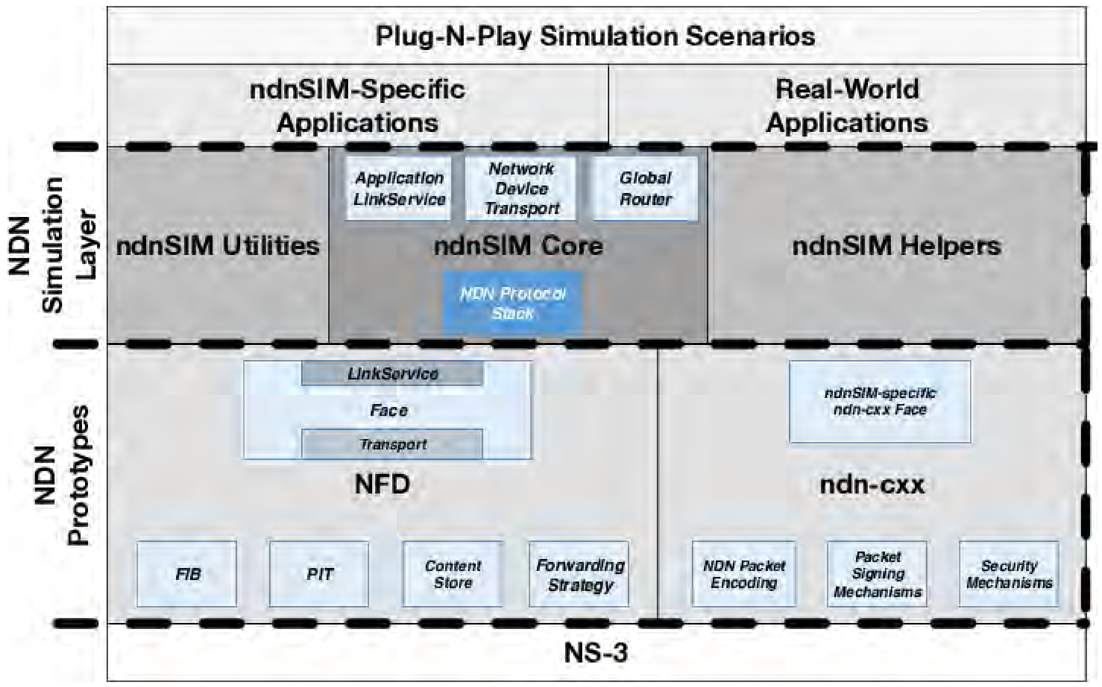


Figure 4.1: NDN Simulation Components [MAZ17]

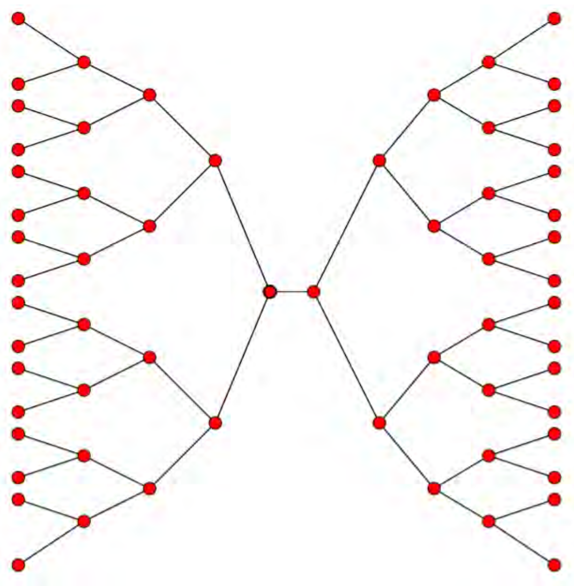
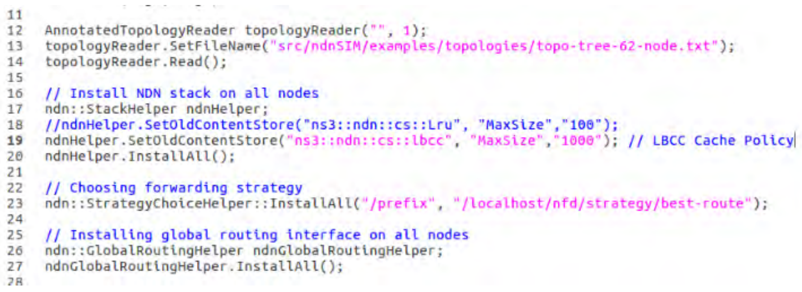


Figure 4.2: A sample NDN topology.

Content Store Module and Customization:
Figure 4.3 shows the customization of NDN parameters. Our customized policy LBCC is shown in Figure 4.4, where we show the topology and policy settings. Here, we define caching policy as well as define the cache size. To show the impact of cache size we change the value of this field.

ndnSIM: NDN simulation module
Content Store

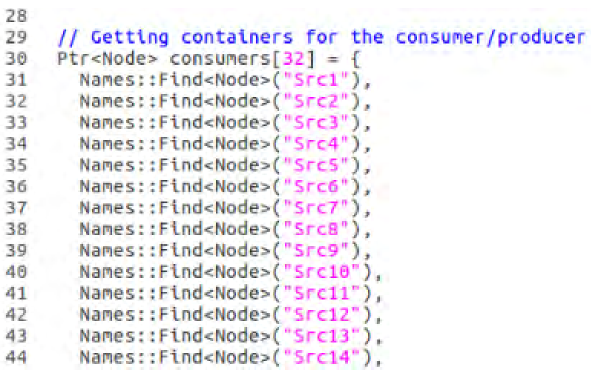
Figure 4.3: CS Module in NDN



```

AnnotatedTopologyReader topologyReader("", 1);
topologyReader.SetFileName("src/ndnSIM/examples/topologies/topo-tree-62-node.txt");
topologyReader.Read();

// Install NDN stack on all nodes
ndn::StackHelper ndnHelper;
//ndnHelper.SetOldContentStore("ns3::ndn::cs::Lru", "MaxSize","100");
ndnHelper.SetOldContentStore("ns3::ndn::cs::lbcc", "MaxSize","1000"); // LBCC Cache Policy
ndnHelper.InstallAll();

// Choosing forwarding strategy
ndn::StrategyChoiceHelper::InstallAll("/prefix", "/localhost/nfd/strategy/best-route");

// Installing global routing interface on all nodes
ndn::GlobalRoutingHelper ndnGlobalRoutingHelper;
ndnGlobalRoutingHelper.InstallAll();

```

Figure 4.4: Cache Policy

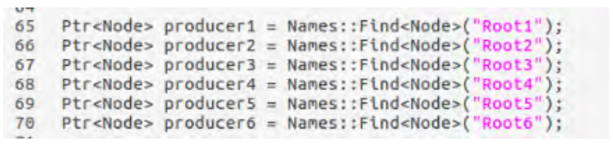
Figure 4.5 shows the consumer declaration in our simulation model.



```

// Getting containers for the consumer/producer
Ptr<Node> consumers[32] = {
  Names::Find<Node>("Src1"),
  Names::Find<Node>("Src2"),
  Names::Find<Node>("Src3"),
  Names::Find<Node>("Src4"),
  Names::Find<Node>("Src5"),
  Names::Find<Node>("Src6"),
  Names::Find<Node>("Src7"),
  Names::Find<Node>("Src8"),
  Names::Find<Node>("Src9"),
  Names::Find<Node>("Src10"),
  Names::Find<Node>("Src11"),
  Names::Find<Node>("Src12"),
  Names::Find<Node>("Src13"),
  Names::Find<Node>("Src14"),
```

Figure 4.5: Consumers

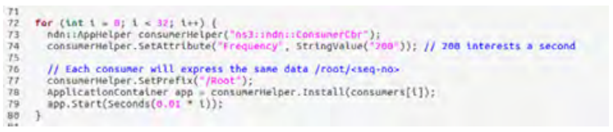
Figure 4.6 shows the service provider declaration.



```
Ptr<Node> producer1 = Names::Find<Node>("Root1");
Ptr<Node> producer2 = Names::Find<Node>("Root2");
Ptr<Node> producer3 = Names::Find<Node>("Root3");
Ptr<Node> producer4 = Names::Find<Node>("Root4");
Ptr<Node> producer5 = Names::Find<Node>("Root5");
Ptr<Node> producer6 = Names::Find<Node>("Root6");
```

Figure 4.6: Providers

In figure Figure 4.7 we define interest rate as 200 per second.



```

for (int i = 0; i < 32; i++) {
  ndn::AppHelper consumerHelper("ns3::ndn::ConsumerCbr");
  consumerHelper.SetAttribute("Frequency", StringValue("200")); // 200 interests a second

  // Each consumer will express the same data /root/<seq-no>
  consumerHelper.SetPrefix("/Root");
  ApplicationContainer app = consumerHelper.Install(consumers[i]);
  app.Start(Seconds(0.01 * i));
}
```

Figure 4.7: Consumer Installation

4.1.1 Simulation Parameters

Now, we provide the simulation parameters used for our simulations in Table 4.1.

Table 4.1: Simulation Parameters

Parameter	**Value**
Packet size	1024 byte
Number of nodes	62
Number of contents	1000
Request Rate	200 requests/sec
Producers	6
Consumers	32
Routers	24

The request arrival rate at each content requester follows *Poisson* arrival process. The simulation time is set to 1000s. Simulations go through a warm-up phase where the content popularity, trends calculation are completed and the information is distributed.

4.1.2 Metrics

We now describe the performance metrics as follows. Three metrics are used to measure the performance of our algorithms. These are *Cache Hit Ratio*, *Latency*, and *Interest Satisfaction Ratio*. We give the details of the metrics as follows.

- *Cache hit ratio* measures the portion of content requests served by a cache. And a higher cache hit ratio means that the contents have been found which ultimately reduces the servers load.

- *Interest satisfaction ratio* implies the number of interests satisfied. Higher interest satisfaction ratio indicates the better performance of NDN.

- *Latency or response time* reflects the delay in getting content, which is the intuitive metric to compute the network performance.

Now, we check the impact on *cache hit ratio* of by varying the following attributes:

- Cache size

- Number of interests

- Experiment run time

We simulate by varying the number of interests and the data load. Besides, we also looked into the impact of number of nodes, interest satisfaction rate and time delay. In the following subsections, we have added the simulation results in graphical form. The details and the raw data for simulation are reported in Appendix. We check the impact on *interest satisfaction ratio* by changing the following parameters.

- Number of interests

We check the impact on the average *response time* by varying the following attribute:

- Cache size

4.1.3 Impact of CS size on Cache Hit Ratio

To measure cache hit ratio we maintain the number of interests to 200 per second, packet size 1024 byte. We vary cache size in each iteration to collect the trace data from 100 kbit, 200 kbit, 300 kbit, 400 kbit, 500 kbit and 600 kbit of CS. When cache size is increased from 100 kbits, the cache hit ratio is increased by 31.75% and simultaneously after increasing CS size 600 kbits, cache hit ratio increases to 50%. If cache size gets better, hit ratio gets better because more contents can be cached and found easily. Figure 4.8 shows the graphical representation of cache hit ratio vs. CS size.

Compared with the performance of our proposed method, we select two other caching strategies CPRL and LRU. When the content store (CS) size is 100 kbit, 200 kbit, 300 kbit, the cache hit rate of LBCC looks almost same as CPRL strategy. But when CS size is increased from 400 kbits to 600 kbits, cache hit ratio is increased compared to CPRL and LRU. If CS increases, cache hit ratio increases because more contents can be cached. The LRU policy actively caches the content block without any selection, which may causes cache redundancy through the network, thereby decreasing the cache hit ratio of the network.
The cache hit ratio in LBCC is directly proportional to Request Trend, encouraging that the competitive advantage of LBCC comes from the areas where users share popular content through caching networks. We use request trend algorithm to calculate content popularity in our LBCC strategy, but in CPRL strategy author use zipf distribution to calculate content popularity technique. In LBCC, we cache content by selecting the important node by using node selection algorithm, but in CPRL, popularity based caching is used.

4.1.4 Impact of the Number of Nodes on Cache Hit Ratio

We keep the cache Size 1000 Kbit, packet size 1024 byte and we set the number of router nodes to 10,20,30,40, and 50. When the node number is 10 we find the cache hit ratio is 30 % and

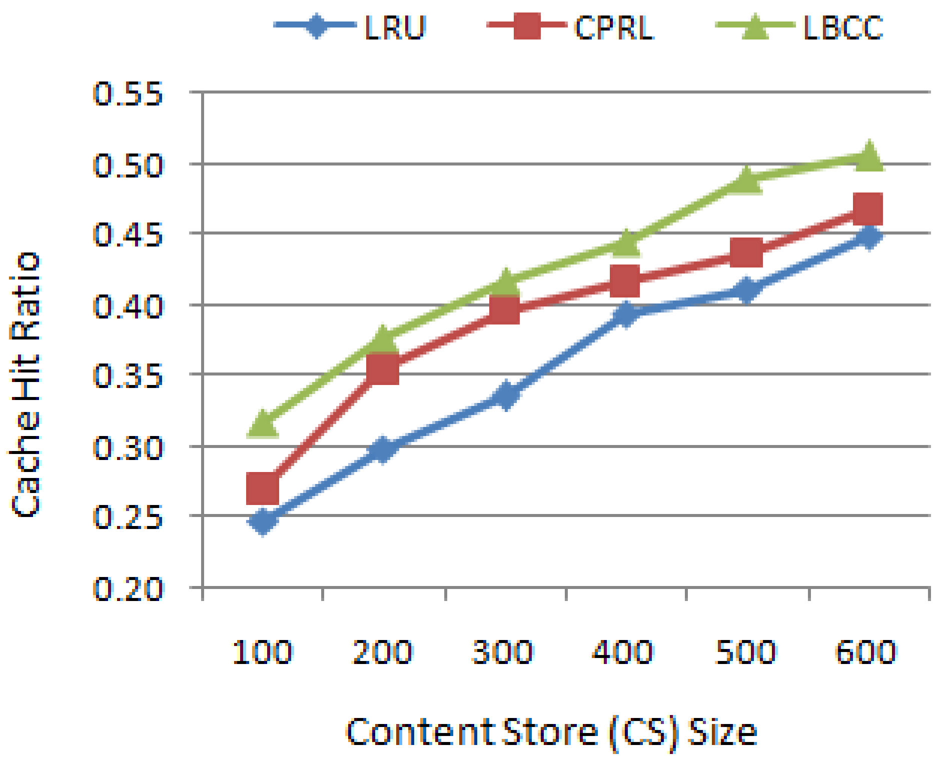


Figure 4.8: Cache Hit Ratio vs. Content Store Size

when the node number is increased to 50, the cache hit ratio is also increased to 55%. Figure 4.9 shows the graphical representation of the number of nodes on cache hit ratio.

We compare the performance of our proposed method with other caching strategies CPRL and LRU also. Cache hit ratio is increased with the increase in the number of nodes. Simulation shows proposed LBCC is better than CPRL and LRU. We use request trend for content popularity calculating that provide best cache hit ratio for LBCC than CPRL and LRU.

4.1.5 Impact of Run Time on Cache Hit Ratio

We keep the cache size to 1000 Kbit, number of interests 200 per second and packet size 1024 byte. We record results in 5s,10s,15s 20s, 25s and 30s. Figure 4.10 is a comparison of the cache hit ratio of the proposed strategy and the CPRL and LRU strategy. When the running time is 5s, the cache hit rate of proposed strategy is lower than that of CPRL strategy. With the increasing of time, the content popularity is increasing, and the cache hit ratio of proposed strategy is higher than that of the existing CPRL and LRU strategy. Simulation results show that our method plays better role in improving the cache hit ratio of NDN, which satisfies our original intention of designing this strategy. The results denote that in LBCC, packet delivery is increased (smaller delay and higher availability). Due to higher hit ratio, more requests are served within network domains. Thus the traffic in backbone network and the workload of content servers are reduced.

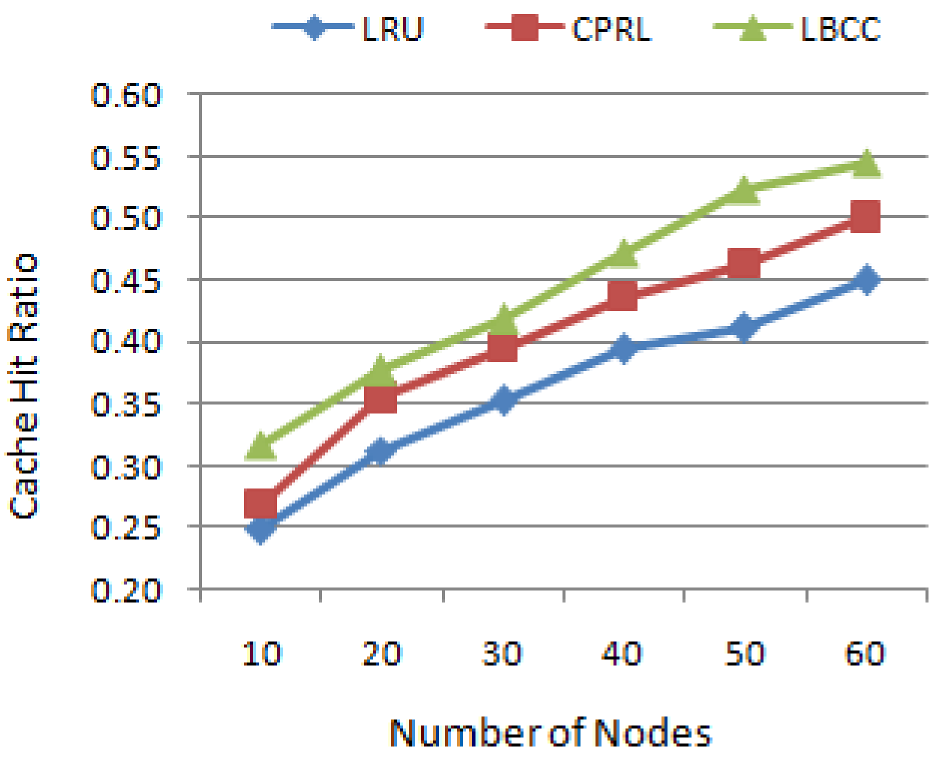


Figure 4.9: Cache Hit Ratio vs. Number of Nodes

Our proposed LBCC uses CRT table on edge nodes to calculate content popularity, so that all other nodes are free from all other computation costs. That is why run time becomes better than the run time of CPRL and LRU strategy.

4.1.6 Impact on Interest Satisfaction Ratio

We keep the number of interests per second 200 and packet size 1024 byte. Figure 4.11 shows better result compared with existing CPRL and LRU strategy.

We compare the performance of LBCC method with CPRL and LRU by varying the number of interests. The number of interests satisfaction ratio is decreased with the increase of the number of interests per second which is natural because increased load decreases service quality. NDN has a mechanism to monitor traffic when multiple users access to the same or different interests, which may lead the PIT overflow. As a result, the delay is increased. Our proposed method is to adjust incoming Interest packet in the early phase of occurrence to propose possible response decisions to realize PIT overflow recovery. We use time duration parameter to calculate the interest lifetime for enhancing the management operations of PIT. For this reason, PIT is not overloaded and data packet is found in less delay. Users will frequently need for popular content rather than unpopular content. Caching popular content on the nodes closer to the users will greatly reduce the userś request latency. Our time duration parameter cache most popular content

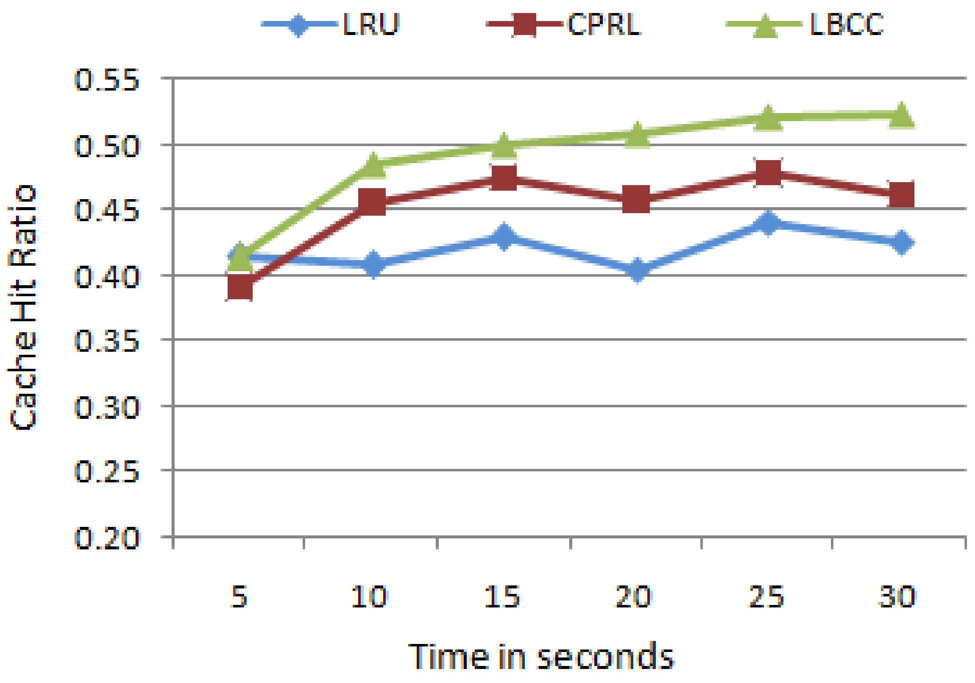


Figure 4.10: Cache Hit Ratio vs Running Time

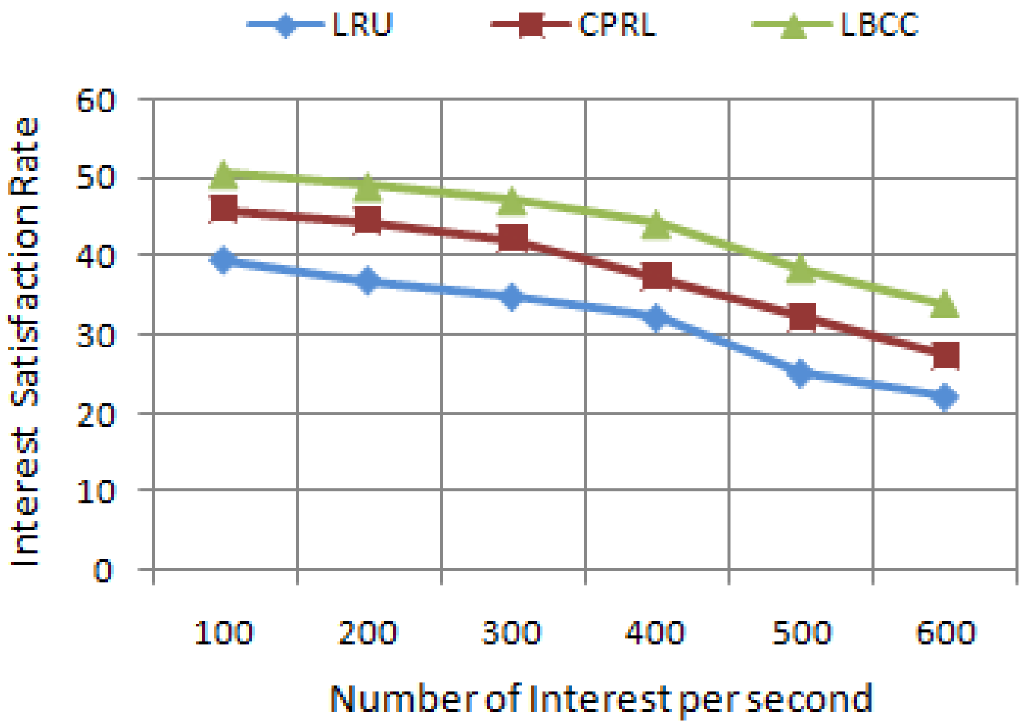


Figure 4.11: Interest Satisfaction Rate vs No. of Interest Per Second

in large time and less popular content cache for short time.

4.1.7 Impact of Number of Nodes on Interest Satisfaction Rate

We keep the cache Size 1000 Kbit, number of interest request 200/s and packet size 1024 byte. Figure 4.12 is a comparison of the interest satisfaction rate among the proposed LBCC strategy and the CPRL and LRU strategy. We set the number of nodes 10, 20, 30, 40, 50 and 60 in our simulator. The graphical result of simulator shows that with the increase of node number, interest satisfaction rate is increased accordingly. When the node number is 10 proposed strategy shows almost same result as existing CPRL strategy. But increasing the node number our proposed strategy shows better result over the existing two. We use a new factor trend to calculate content popularity where content is filtered before caching and for this reason, unique content is cached on each router. Since unique content is cache on all router so we cache large number of content in our system and finally to find interest satisfaction rate is high on our proposed LBCC method.

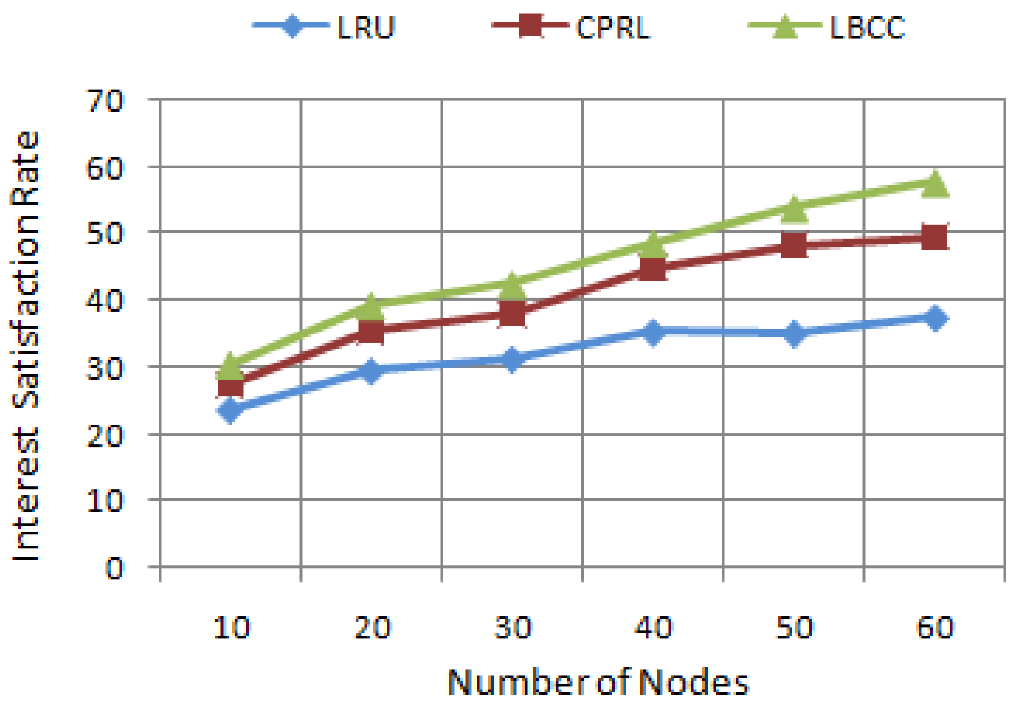


Figure 4.12: Interest Satisfaction Rate vs. Number of Nodes

4.1.8 Impact on the Average Response Time (CS 100 to 600)

We keep the cache Size 1000 Kbit, number of interest per second 200 and packet size 1024 byte. Figure 4.13 is a comparison of the average response time per second of the proposed strategy and the CPRL and LRU strategy. When the cache size is low, the response time or delay is high. In our simulation study, when cache size is 100 kbit, the average response time is low in existing CPRL. With the increase of cache size our proposed LBCC shows better result compared to

CPRL and LRU. The LBCC has a good advantage in terms of average response time and it reduces the average response time, because of LBCC caches most popular content in routers who is closer to the users, while LCE strategy caches all contents in closest routers where the popular contents may be frequently replaced.

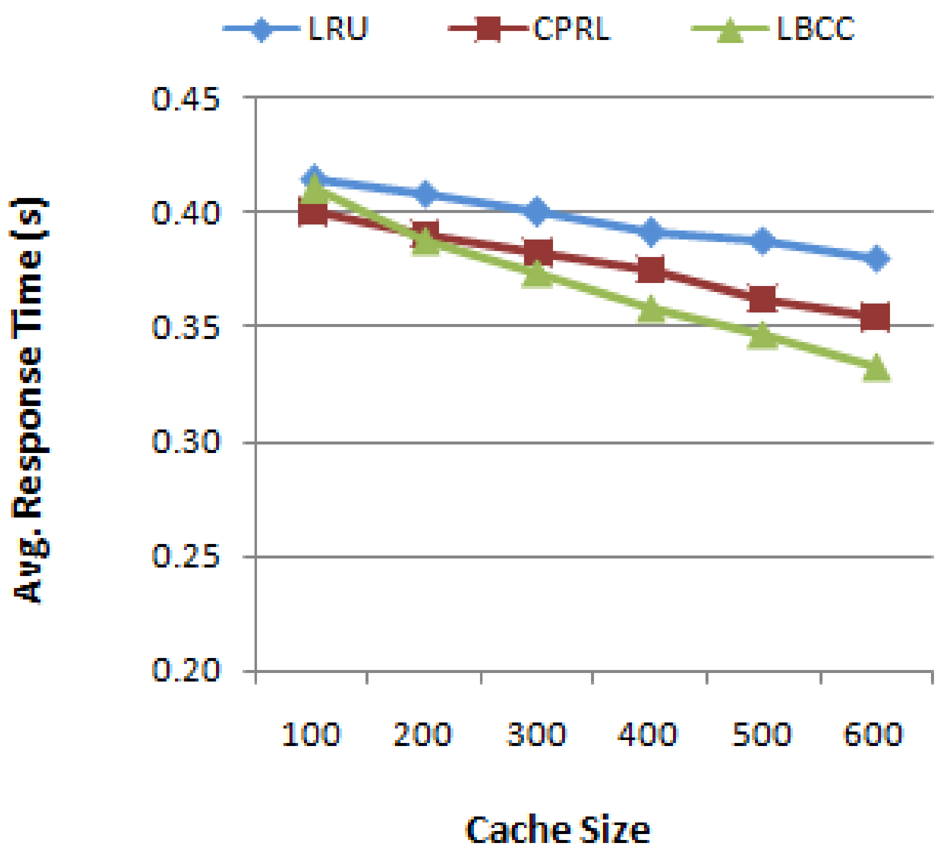


Figure 4.13: Average Response Time(s) vs Cache Size

4.1.9 Impact of Cache Size on the Average Response Time (CS 1000 to 6000)

Increasing CS size 1000 to 6000 Kbit and it is found that cache hit ratio is increased compared to CPRL and LRU. But when the CS size is increased from 1000 to 6000, Avg. response time is increased compared to CPRL. It should be noted that, our strategy gives better response time for cache size 1000. It may need extra computation cost for searching the requested contents.

Figure 4.14 and Figure 4.15 shows the impact of large scale data.

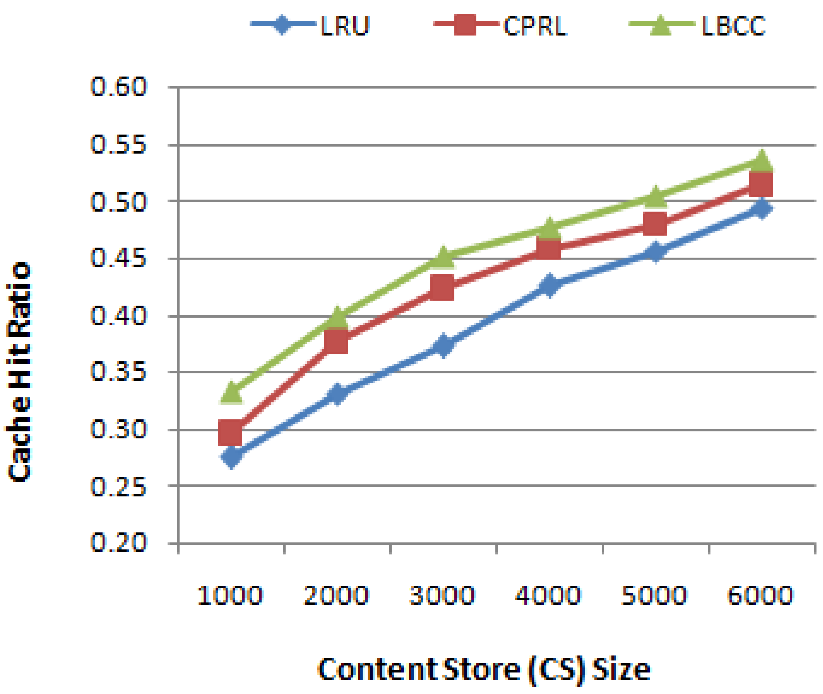


Figure 4.14: Cache Hit Ratio vs Cache Size

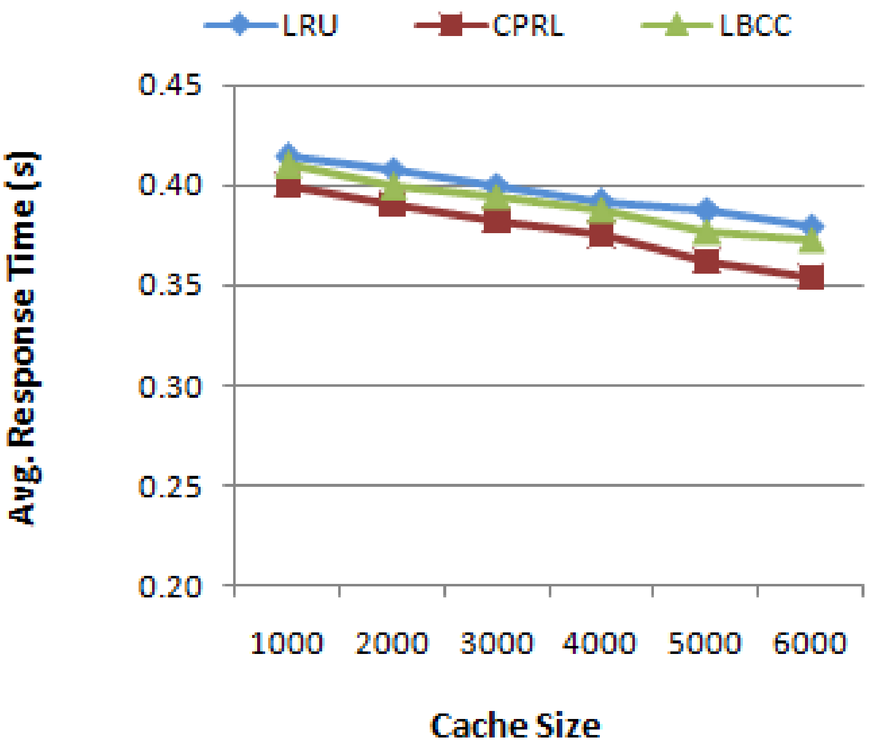


Figure 4.15: Avg. Response Time vs Cache Size

Chapter 5

Conclusion

In-network caching offers quick response and easy packet access in Name Data Network (NDN). Since in-network caching operates in a distributed environment, challenges exist in designing an efcient in-network caching for a large-scale network. In this research, we provide a hybrid framework content caching strategy. Our strategy jointly considers the content popularity for caching node selection process for better outcome on caching strategy. To overcome the data redundancy, we use request trend parameter to calculate content popularity. We select caching nod e for the requested contents. Our method is distributed and scalable. Our algorithm is suitable for any network topology.

We summarize our contribution in this research as follows.

- We have studied several content caching strategies, such as probability based, popularity based and age based strategy for caching content in NDN.
- We have proposed a content caching method based on content popularity technique where two new parameters have been. Our newly designed parameters are Request Rate and Request Trend to calculate content popularity in Named Data Network. After calculating the content popularity we select the router node where to cache the popular content. For selecting router position, we use hop counting and popularity ranking technique.
- We validate our method providing simulation based results using standard NDN simulator ndnSIM.

.

5.1 Future Improvement

We will extend this research to handle other network parameters. We will find the behaviour pattern by using exible update intervals. For that we will use sliding window instead of static

update period, which will make caching decision more accurately.
In future, we try to use a dynamic popularity-based caching permission strategy which will take advantages of dynamic nature of interest packets. It will facilitate in that way that, on-path routers can obtain the information about the content popularity and use dynamic popularity threshold to make cache permission plans.
We will explore the impact of cache control flag to tune the redundancy.

References

[AMZ12] Alexander Afanasyev, Ilya Moiseenko, and Lixia Zhang. ndnsim: Ndn simulator for ns-3. *Department of Computer Science, University of California at Los Angeles, CA, USA,*, NDN-0005, 2012.

[CHPP13] Wei Koong Chai, Diliang He, Ioannis Psaras, and George Pavlou. Cache less for more in information-centric networks (extended version). In *Computer Communication*, volume 36, pages 758–770, 2013.

[DHL+12] Jie Dai, Zhan Hu, Bo Li, Jiangchuan Liu, and Baochun Li. Collaborative hierarchical caching with dynamic request routing for massive content distribution. In *IEEE INFOCOM*, pages 2444–2452. IEEE, 2012.

[JXY14] Jia Ji, Mingwei Xu, and Yuan Yang. Content-hierarchical intra-domain cooperative caching for information-centric networks. In *8th International Conference on Future Internet Technology*. IEEE, 2014.

[kay] Named data networking. `http://named-data.net/`. Last Accessed: 2019-02-02.

[LWL+12] Jun Li, Hao Wu, Bin Liu, et al. Popularity-driven coordinated caching in named data networking. In *8th ACM/IEEE Symposium. on Architecture, Network and Communication Systems (ANCS)*, pages 15–26. IEEE, 2012.

[MAIZ15] Spyridon Mastorakis, Alexander Afanasyev, Moiseenko Ilya, and Lixia Zhang. ndnsim 2.0: A new version of the ndn simulator for ns-3. In *NDN, Technical Report NDN-0028,*, 2015.

[MAZ17] Spyridon Mastorakis, Alexander Afanasyev, and Lixia Zhang. On the evolution of ndnsim: an open-source simulator for ndn experimentation. In *ACM SIGCOMM Computer Communication Review, Volume 47 Issue 3*, pages 19–33. IEEE, July 2017.

[MXW12] Zhongxing Ming, Mingwei Xu, and Dan Wang. Age-based cooperative caching in information-centric networks,. In *IEEE International Conference on Computer*

Communications (INFOCOM WKSHPS), pages 268–273. INFOCOM WKSHPS, March 2012.

[PCP12] Ioannis Psaras, Wei Koong Chai, and George Pavlou. Probabilistic in-network caching for information-centric networks. In *ACM Workshop on Information-Centric Networking (ICN)*, pages 55–60. ICN, 2012.

[RQ$^+$14] Jing Ren, Wen Qi, et al. Magic: A distributed max-gain in-network caching strategy in information-centric networks. In *IEEE Conference on Computer Communication. Workshops (INFOCOM WKSHPS)*, pages 470–475. INFOCOM WKSHPS, April 2014.

[SNRJE18] Hamonangan Situmorang, Syambas Nana Rachmana, Tutun Juhana, and Ian Edward. A simulation of cache replacement strategy on named data network. In *12th International Conference on Telecommunication Systems, Services, and Applications (TSSA)*. IEEE, 2018.

[VLSY15] Athanasios V Vasilakos, Zhe Li, Gwendal Simon, and Wei You. Information centric network: Research challenges and opportunities. *Journal of Network and Computer Applications*, 52:1–10, 2015.

[WSG$^+$14] Wei Wang, Yi Sun, Yang Guo, et al. Crcache: Exploiting the correlation between content popularity and network topology information for icn caching. In *IEEE International Conference on Computer Communication (INFOCOM)*, pages 3191–3196. IEEE, June 2014.

[WXF13] Yu Wang, Mingwei Xu, and Zhen Feng. Hop-based probabilistic caching for information-centric networks. In *IEEE Global Communication Conference. (GLOBECOM)*, pages 2102–2107. IEEE, December 2013.

[WZB13] Jason Min Wang, Jun Zhung, and Brahim Bensaou. Intra-as cooperative caching for content-centric networks,. In *3rd ACM SIGCOMM Workshop on Information-Centric Networking (ICN)*, pages 61–66. ICN, 2013.

[YGS16] Huan Yan, Deyun Gao, and Wei Su. A hierarchical cluster-based caching for named data networking. In *IEEE/CIC International Conference on Communications in China (ICCC)*, pages 1–6. IEEE, 2016.

[YL18] Meiju Yu and Ru Li. Dynamic popularity-based caching permission strategy for named data networking. In *2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design ((CSCWD))*. IEEE, 2018.

[YLLL17] Meiju Yu, Ru Li, Yinggi Liu, and Yinggi Li. A caching strategy based on content popularity and router level for ndn. In *7th IEEE International Conference on Electronics Information and Emergency Communication (ICEIEC)*. IEEE, 2017.

[ZWWQ17] Xiaodong Zhu, Jinlin Wang, Lingfang Wang, and Weining Qi. Popularity-based neighborhood collaborative caching for information-centric networks. In *IEEE 36th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2017.

Appendix A

Raw Data

A.1 Cache Hit Ratio vs Content Store(CS) Size

Table A.1: Cache Hit Ratio vs Content Store(CS) Size(LBCC).

Content Store(CS) Size	**Cache Hit Ratio**
100	0.3175
200	0.37634
300	0.41760
400	0.44536
500	0.48974
600	0.50551

Table A.2: Cache Hit Ratio vs Content Store(CS) Size (CPRL).

Content Store(CS) Size	**Cache Hit Ratio**
100	0.26925
200	0.354838
300	0.395598
400	0.416104
500	0.435967
600	0.467451

Table A.3: Cache Hit Ratio vs Content Store(CS) Size (LRU).

Content Store Size	Cache Hit Ratio
100	0.24750
200	0.29782
300	0.33533
400	0.39385
500	0.41071
600	0.4494413

A.2 Cache Hit Ratio vs Number of Nodes

Table A.4: Cache Hit Ratio vs Number of Nodes (LBCC).

Number of Nodes	Cache Hit Ratio
10	0.3175
20	0.37634
30	0.41760
40	0.472083021
50	0.524027014
60	0.545948923

Table A.5: Cache Hit Ratio vs Number of Nodes(CPRL).

Number of Nodes	Cache Hit Ratio
10	0.26925
20	0.354838
30	0.395598
40	0.436909227
50	0.462126063
60	0.500172759

Table A.6: Cache Hit Ratio vs Number of Nodes(LRU).

Number of Nodes	Cache Hit Ratio
10	0.24750
20	0.312715679
30	0.33533
40	0.39385
50	0.41071
60	0.4494413

A.3 Interest Satisfaction Rate vs Number of Interest Per Second

Table A.7: Interest Satisfaction Rate vs No. of Interest Per Second(LBCC)

No. of Interest Per Second	**Interest Satisfaction Rate**
100	50.5508
200	48.9745
300	47.1618
400	44.1860
500	38.4846
600	33.975

Table A.8: Interest Satisfaction Rate vs No. of Interest Per Second(CPRL)

No. of Interest Per Second	**Interest Satisfaction Rate**
100	45.943915
200	44.497248
300	42.2105526
400	37.45936484
500	32.25806452
600	27.5

Table A.9: Interest Satisfaction Rate vs No. of Interest Per Second(LRU)

No. of Interest Per Second	**Interest Satisfaction Rate**
100	39.48422634
200	36.89344672
300	34.78369592
400	32.10802701
500	25.10627657
600	22.075

A.4 Cache Hit Ratio vs Running Time

Table A.10: Cache Hit Ratio vs Running Time(LBCC)

Running Time	**Cache Hit Ratio**
5	0.414
10	0.485
15	0.5
20	0.508
25	0.521
30	0.523

Table A.11: Cache Hit Ratio vs Running Time(CPRL)

Running Time	**Cache Hit Ratio**
5	0.391
10	0.456
15	0.474
20	0.458
25	0.478
30	0.462

Table A.12: Cache Hit Ratio vs Running Time(LRU)

Running Time	**Cache Hit Ratio**
5	0.444
10	0.408
15	0.429
20	0.405
25	0.441
30	0.426

A.5 Interest Satisfaction Rate vs Number of Nodes

Table A.13: Interest Satisfaction Rate vs No. of Nodes (LBCC)

No. of Nodes	Interest Satisfaction Rate
10	30.33049574
20	39.17958979
30	42.4456114
40	48.60465116
50	53.87846962
60	57.7575

Table A.14: Interest Satisfaction Rate vs No. of Nodes (CPRL)

No. of Nodes	Interest Satisfaction Rate
10	27.56634952
20	35.5977989
30	37.98949737
40	44.95123781
50	48.38709677
60	49.5

Table A.15: Interest Satisfaction Rate vs No. of Nodes (LRU)

No. of Nodes	Interest Satisfaction Rate
10	23.6905358
20	29.51475738
30	31.30532633
40	35.31882971
50	35.1487872
60	37.5275

A.6 Average Response Time(s) vs Cache Size

Table A.16: Average Response Time(s) vs Cache Size (LBCC).

Cache Size	Time
100	0.41
200	0.388
300	0.374
400	0.354
500	0.347
600	0.3323

Table A.17: Average Response Time(s) vs Cache Size (CPRL).

Cache Size	Time
100	0.4
200	0.39
300	0.3828
400	0.3745
500	0.362
600	0.354

Table A.18: Average Response Time(s) vs Cache Size (LRU).

Cache Size	Time
100	0.414
200	0.408
300	0.4
400	0.391
500	0.388
600	0.3812

Appendix B

Codes

B.1 NDN Content Store

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2014-2018,  Regents of the University of California,
 *                           Arizona Board of Regents,
 *                           Colorado State University,
 *                           University Pierre & Marie Curie, Sorbonne Un
 *                           Washington University in St. Louis,
 *                           Beijing Institute of Technology,
 *                           The University of Memphis.
 *
 * This file is part of NFD (Named Data Networking Forwarding Daemon).
 * See AUTHORS.md for complete list of NFD authors and contributors.
 *
 * NFD is free software: you can redistribute it and/or modify it under t
 * of the GNU General Public License as published by the Free Software Fo
 * either version 3 of the License, or (at your option) any later version
 *
 * NFD is distributed in the hope that it will be useful, but WITHOUT ANY
 * without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License alon
 * NFD, e.g., in COPYING.md file.  If not, see <http://www.gnu.org/licens
 */
```

```

#include "cs.hpp"
#include "core/algorithm.hpp"
#include "core/logger.hpp"

#include <ndn-cxx/lp/tags.hpp>
#include <ndn-cxx/util/concepts.hpp>

namespace nfd {
namespace cs {

NDN_CXX_ASSERT_FORWARD_ITERATOR(Cs::const_iterator);

NFD_LOG_INIT("ContentStore");

unique_ptr<Policy>
makeDefaultPolicy()
{
  //const std::string DEFAULT_POLICY = "lfu";
  //const std::string DEFAULT_POLICY = "cprl";
  const std::string DEFAULT_POLICY = "lbcc";
  return Policy::create(DEFAULT_POLICY);
}

Cs::Cs(size_t nMaxPackets)
  : m_shouldAdmit(true)
  , m_shouldServe(true)
{
  this->setPolicyImpl(makeDefaultPolicy());
  m_policy->setLimit(nMaxPackets);
}

void
Cs::insert(const Data& data, bool isUnsolicited)
{
  if (!m_shouldAdmit || m_policy->getLimit() == 0) {
    return;
  }
  NFD_LOG_DEBUG("insert␣" << data.getName());

  // recognize CachePolicy
  shared_ptr<lp::CachePolicyTag> tag = data.getTag<lp::CachePolicyTag>();
  if (tag != nullptr) {
    lp::CachePolicyType policy = tag->get().getPolicy();
    if (policy == lp::CachePolicyType::NO_CACHE) {
      return;
    }
  }

  iterator it;
  bool isNewEntry = false;
  std::tie(it, isNewEntry) = m_table.emplace(data.shared_from_this(), isU
  EntryImpl& entry = const_cast<EntryImpl&>(*it);

  entry.updateStaleTime();

  if (!isNewEntry) { // existing entry
    // XXX This doesn't forbid unsolicited Data from refreshing a solicit
    if (entry.isUnsolicited() && !isUnsolicited) {
      entry.unsetUnsolicited();
    }

    m_policy->afterRefresh(it);
  }
  else {
    m_policy->afterInsert(it);
  }
}

void
Cs::erase(const Name& prefix, size_t limit, const AfterEraseCallback& cb)
{
  BOOST_ASSERT(static_cast<bool>(cb));

  iterator first = m_table.lower_bound(prefix);
  iterator last = m_table.end();
  if (prefix.size() > 0) {
    last = m_table.lower_bound(prefix.getSuccessor());
  }

  size_t nErased = 0;
  while (first != last && nErased < limit) {
    m_policy->beforeErase(first);
    first = m_table.erase(first);
    ++nErased;
  }

  if (cb) {
    cb(nErased);
  }
}

void
Cs::find(const Interest& interest,
         const HitCallback& hitCallback,
         const MissCallback& missCallback) const
{
  BOOST_ASSERT(static_cast<bool>(hitCallback));
  BOOST_ASSERT(static_cast<bool>(missCallback));

  if (!m_shouldServe || m_policy->getLimit() == 0) {
    missCallback(interest);
    return;
  }
  const Name& prefix = interest.getName();
  bool isRightmost = interest.getChildSelector() == 1;
  NFD_LOG_DEBUG("find␣" << prefix << (isRightmost ? "␣R" : "␣L"));

  iterator first = m_table.lower_bound(prefix);
  iterator last = m_table.end();
  if (prefix.size() > 0) {
    last = m_table.lower_bound(prefix.getSuccessor());
  }

  iterator match = last;
  if (isRightmost) {
    match = this->findRightmost(interest, first, last);
  }
  else {
    match = this->findLeftmost(interest, first, last);
  }

  if (match == last) {
    NFD_LOG_DEBUG("␣␣no-match");
    missCallback(interest);
    return;
  }
  NFD_LOG_DEBUG("␣␣matching␣" << match->getName());
  m_policy->beforeUse(match);
  hitCallback(interest, match->getData());
}

iterator
Cs::findLeftmost(const Interest& interest, iterator first, iterator last)
{
  return std::find_if(first, last, bind(&cs::EntryImpl::canSatisfy, _1, i
}

iterator
Cs::findRightmost(const Interest& interest, iterator first, iterator last
{
  // Each loop visits a sub-namespace under a prefix one component longer
  // If there is a match in that sub-namespace, the leftmost match is ret
  // otherwise, loop continues.

  size_t interestNameLength = interest.getName().size();
  for (iterator right = last; right != first;) {
    iterator prev = std::prev(right);

    // special case: [first,prev] have exact Names
    if (prev->getName().size() == interestNameLength) {
      NFD_LOG_TRACE("␣␣find-among-exact␣" << prev->getName());
      iterator matchExact = this->findRightmostAmongExact(interest, first
      return matchExact == right ? last : matchExact;
    }

    Name prefix = prev->getName().getPrefix(interestNameLength + 1);
    iterator left = m_table.lower_bound(prefix);

    // normal case: [left,right) are under one-component-longer prefix
    NFD_LOG_TRACE("␣␣find-under-prefix␣" << prefix);
    iterator match = this->findLeftmost(interest, left, right);
    if (match != right) {
      return match;
    }
    right = left;
  }
  return last;
}

iterator
Cs::findRightmostAmongExact(const Interest& interest, iterator first, ite
{
  return find_last_if(first, last, bind(&EntryImpl::canSatisfy, _1, inter
}

void
Cs::dump()
{
  NFD_LOG_DEBUG("dump␣table");
  for (const EntryImpl& entry : m_table) {
    NFD_LOG_TRACE(entry.getFullName());
  }
}

void
Cs::setPolicy(unique_ptr<Policy> policy)
{
  BOOST_ASSERT(policy != nullptr);
  BOOST_ASSERT(m_policy != nullptr);
  size_t limit = m_policy->getLimit();
  this->setPolicyImpl(std::move(policy));
  m_policy->setLimit(limit);
}

void
Cs::setPolicyImpl(unique_ptr<Policy> policy)
{
  NFD_LOG_DEBUG("set-policy␣" << policy->getName());
  m_policy = std::move(policy);
  m_beforeEvictConnection = m_policy->beforeEvict.connect([this] (iterato
      m_table.erase(it);
    });

  m_policy->setCs(this);
  BOOST_ASSERT(m_policy->getCs() == this);
}

void
Cs::enableAdmit(bool shouldAdmit)
{
  if (m_shouldAdmit == shouldAdmit) {
    return;
  }
  m_shouldAdmit = shouldAdmit;
  NFD_LOG_INFO((shouldAdmit ? "Enabling" : "Disabling") << "␣Data␣admitta
}

void
Cs::enableServe(bool shouldServe)
{
  if (m_shouldServe == shouldServe) {
    return;
  }
  m_shouldServe = shouldServe;
  NFD_LOG_INFO((shouldServe ? "Enabling" : "Disabling") << "␣Data␣serving
}

} // namespace cs
} // namespace nfd
```

B.2 NDN Content Store Header File

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2014-2018,  Regents of the University of California,
 *                           Arizona Board of Regents,
 *                           Colorado State University,
 *                           University Pierre & Marie Curie, Sorbonne Un
 *                           Washington University in St. Louis,
 *                           Beijing Institute of Technology,
 *                           The University of Memphis.
 *
 * This file is part of NFD (Named Data Networking Forwarding Daemon).
 * See AUTHORS.md for complete list of NFD authors and contributors.
 *
 * NFD is free software: you can redistribute it and/or modify it under t
 * of the GNU General Public License as published by the Free Software Fo
 * either version 3 of the License, or (at your option) any later version
 *
 * NFD is distributed in the hope that it will be useful, but WITHOUT ANY
 * without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License alon
 * NFD, e.g., in COPYING.md file.  If not, see <http://www.gnu.org/licens
 */

#ifndef NFD_DAEMON_TABLE_CS_HPP
#define NFD_DAEMON_TABLE_CS_HPP

#include "cs-policy.hpp"
#include "cs-internal.hpp"
#include "cs-entry-impl.hpp"
#include <ndn-cxx/util/signal.hpp>
#include <boost/iterator/transform_iterator.hpp>
```

```

namespace nfd {
namespace cs {

/** \brief implements the Content Store
 *
 *  This Content Store implementation consists of a Table and a replaceme
 *
 *  The Table is a container ( \c std::set ) sorted by full Names of stor
 *  Data packets are wrapped in Entry objects. Each Entry contains the Da
 *  and a few additional attributes such as when the Data becomes non-fre
 *
 *  The replacement policy is implemented in a subclass of \c Policy.
 */
class Cs : noncopyable
{
public:
  explicit
  Cs(size_t nMaxPackets = 10);

  /** \brief inserts a Data packet
   */
  void
  insert(const Data& data, bool isUnsolicited = false);

  using AfterEraseCallback = std::function<void(size_t nErased)>;

  /** \brief asynchronously erases entries under \p prefix
   *  \param prefix name prefix of entries
   *  \param limit max number of entries to erase
   *  \param cb callback to receive the actual number of erased entries;
   *            it may be invoked either before or after erase() returns
   */
  void
  erase(const Name& prefix, size_t limit, const AfterEraseCallback& cb);

  using HitCallback = std::function<void(const Interest&, const Data&)>;
  using MissCallback = std::function<void(const Interest&)>;

  /** \brief finds the best matching Data packet
   *  \param interest the Interest for lookup
   *  \param hitCallback a callback if a match is found; must not be empty
   *  \param missCallback a callback if there's no match; must not be emp
   *  \note A lookup invokes either callback exactly once.
   *        The callback may be invoked either before or after find() ret
   */
  void
  find(const Interest& interest,
       const HitCallback& hitCallback,
       const MissCallback& missCallback) const;

  /** \brief get number of stored packets
   */
  size_t
  size() const
  {
    return m_table.size();
  }

public: // configuration
  /** \brief get capacity (in number of packets)
   */
  size_t
  getLimit() const
  {
    return m_policy->getLimit();
  }

  /** \brief change capacity (in number of packets)
   */
  void
  setLimit(size_t nMaxPackets)
  {
    return m_policy->setLimit(nMaxPackets);
  }

```

```
/** \brief get replacement policy
 */
Policy*
getPolicy() const
{
  return m_policy.get();
}

/** \brief change replacement policy
 *  \pre size() == 0
 */
void
setPolicy(unique_ptr<Policy> policy);

/** \brief get CS_ENABLE_ADMIT flag
 *  \sa https://redmine.named-data.net/projects/nfd/wiki/CsMgmt#Update-
 */
bool
shouldAdmit() const
{
  return m_shouldAdmit;
}

/** \brief set CS_ENABLE_ADMIT flag
 *  \sa https://redmine.named-data.net/projects/nfd/wiki/CsMgmt#Update-
 */
void
enableAdmit(bool shouldAdmit);

/** \brief get CS_ENABLE_SERVE flag
 *  \sa https://redmine.named-data.net/projects/nfd/wiki/CsMgmt#Update-
 */
bool
shouldServe() const
{
  return m_shouldServe;
}

```

```
  /** \brief set CS_ENABLE_SERVE flag
   *  \sa https://redmine.named-data.net/projects/nfd/wiki/CsMgmt#Update-
   */
  void
  enableServe(bool shouldServe);

public: // enumeration
  struct EntryFromEntryImpl
  {
    typedef const Entry& result_type;

    const Entry&
    operator()(const EntryImpl& entry) const
    {
      return entry;
    }
  };

  /** \brief ContentStore iterator (public API)
   */
  typedef boost::transform_iterator<EntryFromEntryImpl, iterator, const E

  const_iterator
  begin() const
  {
    return boost::make_transform_iterator(m_table.begin(), EntryFromEntry
  }

  const_iterator
  end() const
  {
    return boost::make_transform_iterator(m_table.end(), EntryFromEntryIm
  }

private: // find
  /** \brief find leftmost match in [first,last)
   *  \return the leftmost match, or last if not found
   */
  iterator
  findLeftmost(const Interest& interest, iterator left, iterator right) c

  /** \brief find rightmost match in [first,last)
   *  \return the rightmost match, or last if not found
   */
  iterator
  findRightmost(const Interest& interest, iterator first, iterator last)

  /** \brief find rightmost match among entries with exact Names in [firs
   *  \return the rightmost match, or last if not found
   */
  iterator
  findRightmostAmongExact(const Interest& interest, iterator first, itera

  void
  setPolicyImpl(unique_ptr<Policy> policy);

PUBLIC_WITH_TESTS_ELSE_PRIVATE:
  void
  dump();

private:
  Table m_table;
  unique_ptr<Policy> m_policy;
  signal::ScopedConnection m_beforeEvictConnection;

  bool m_shouldAdmit; ///< if false, no Data will be admitted
  bool m_shouldServe; ///< if false, all lookups will miss
};

} // namespace cs

using cs::Cs;

} // namespace nfd

#endif // NFD_DAEMON_TABLE_CS_HPP
```

B.3 Popularity Based LBCC Caching File

```
#include "cs-policy-lbcc.hpp"
#include "cs.hpp"

namespace nfd {
namespace cs {
namespace lbcc {

const std::string LbccPolicy::POLICY_NAME = "lbcc";
NFD_REGISTER_CS_POLICY(LruPolicy);

struct PopularityTable
{
   Ptr<Item> item;
   float popularity;
};

struct order_by_popularity
{
    inline bool operator() (const PopularityTable& t1, const PopularityTa
    {
        return (t1.popularity < t2.popularity);
    }
};


LbccPolicy::LbccPolicy()
  : Policy(POLICY_NAME)
{
}

std::list<PopularityTable> _PopularityTable;

void
LbccPolicy::doAfterInsert(iterator i)
{
  this->insertToQueue(i, true);
  this->evictEntries();
}

void
LbccPolicy::doAfterRefresh(iterator i)
{
  this->insertToQueue(i, false);
}

void
LbccPolicy::doBeforeErase(iterator i)
{
  m_queue.get<1>().erase(i);
}

void
LbccPolicy::doBeforeUse(iterator i)
{
  this->insertToQueue(i, false);
}

void
LbccPolicy::evictEntries()
{
  BOOST_ASSERT(this->getCs() != nullptr);
  while (this->getCs()->size() > this->getLimit()) {
    BOOST_ASSERT(!m_queue.empty());
    iterator i = m_queue.front();
        int position =  calculateContentStorePosition(i);
        if(position == this->getCs().position())
        {
                m_queue.pop_front();
                m_queue.push_back(i);
                this->emitSignal(beforeEvict, i);
        }
  }
}

```

```
iterator
LbccPolicy::calculateRequestTrend(Queue m_queue, Ptr<Item> item)
{
    float _iTrend= 1.00; // Initial Content Trend,
    float _k= 1.00; // Constant, that will used to prioritize recent cont
    float _iTrendValue= 1.00; // Initial Content Trend,

    Ptr<Item> lastRequest = null;
    Ptr<Item> currentRequest = null;

    for(int i=0;i<this->getLimit();i++)
    {
                currentRequest = m_queue.DoPeek(i);

                if(currentRequest != item){
                        continue;
                }

                if(i > 0){
                  lastRequest = m_queue.DoPeek(i - 1);
                }

        if(currentRequest ==  lastRequest)
                {
                   _iTrend = _iTrend + 1;
                }
                else
                {
                   if(_iTrend > 1.00){
                                _iTrendValue +=_iTrend * _k;
                        }
                }
    }
    return _iTrendValue;
}


iterator
```

```
LbccPolicy::calculateTotalTrend(Queue m_queue)
{
    float _iTrend= 1.00; // Initial Content Trend,
    float _k= 1.00; // Constant, that will used to prioritize recent cont
    float _iTrendValue= 1.00; // Initial Content Trend,

    Ptr<Item> lastRequest = null;
    Ptr<Item> currentRequest = null;

    for(int i=0;i<this->getLimit();i++)
    {
            currentRequest = m_queue.DoPeek(i);
                if(i > 0){
                  lastRequest = m_queue.DoPeek(i - 1);
                }

        if(currentRequest ==  lastRequest)
                {
                   _iTrend = _iTrend + 1;
                }
                else
                {
                   if(_iTrend > 1.00){
                                _iTrendValue +=_iTrend * _k;
                        }
                }
    }
    return _iTrendValue;
}


iterator
LbccPolicy::calculateRequestRate(Queue m_queue,Ptr<Item> item)
{
    float _iRequestrate = 0.0;
    for(int i=0;i<this->getLimit();i++)
    {
            currentRequest = m_queue.DoPeek(i);
```

```
                if(currentRequest == item){
                        _iRequestrate = _iRequestrate + 1;
                }
    }
    return _iRequestrate;
}

iterator
LbccPolicy::calculatePopularity(Queue m_queue, Ptr<Item> item)
{
    float _popularity = 0.0;
    float _requestRate = calculateRequestRate(m_queue,item) / m_queue.siz
        float _requestTrend = calculateRequestTrend(m_queue,item) / calcu
        _popularity = (_requestRate + _requestTrend)/2;
    return _popularity;
}

iterator
LbccPolicy::calculateContentStorePosition(Ptr<Item> item){

        std::sort(_PopularityTable.begin(), _PopularityTable.end(), order
        PopularityTable _popularityItem =  std::find(_PopularityTable.beg
        int index = std::distance(_PopularityTable.begin(), _popularityIt
        return index -1;
}

void
LbccPolicy::insertToQueue(iterator i, bool isNewEntry)
{
  Queue::iterator it;
  bool isNew = false;

  // push_back only if iterator i does not exist
  std::tie(it, isNew) = m_queue.push_back(i);

  // check element exist or not in _PopularityTable
   PopularityTable pTable = NULL;

  float _popularity =  calculatePopularity(it,i);
  pTable->popularity = _popularity;
  pTable->item = i;
  // check element exist or not in _PopularityTable
  bool found = (std::find(_PopularityTable.begin(), _PopularityTable.end(
  if(!found){
        _PopularityTable.insert(pTable);
  }else{
         _PopularityTable.erase(std::find(_PopularityTable.begin(), _Popu
         _PopularityTable.insert(pTable);
  }
  BOOST_ASSERT(isNew == isNewEntry);
  if (!isNewEntry) {
    m_queue.relocate(m_queue.end(), it);
  }
}

} // namespace lbcc
} // namespace cs
} // namespace nfd
```

B.4 Popularity Based LBCC Caching Header File

```
#ifndef NFD_DAEMON_TABLE_CS_POLICY_LBCC_HPP
#define NFD_DAEMON_TABLE_CS_POLICY_LBCC_HPP

#include "cs-policy.hpp"

#include <boost/multi_index_container.hpp>
#include <boost/multi_index/sequenced_index.hpp>
#include <boost/multi_index/hashed_index.hpp>

namespace nfd {
namespace cs {
namespace lbcc {

struct EntryItComparator
{
  bool
  operator()(const iterator& a, const iterator& b) const
  {
    return *a < *b;
  }
};

typedef boost::multi_index_container<
    iterator,
    boost::multi_index::indexed_by<
      boost::multi_index::sequenced<>,
      boost::multi_index::ordered_unique<
        boost::multi_index::identity<iterator>, EntryItComparator
      >
    >
  > Queue;

/** \brief Lbcc cs replacement policy
 *
 * In this policy we will calculate the request rate
 * (RR : number of individual request devided by total number of request)
 * well as calculate Request Trend (RT: Block of same request) to identif
 */
class LbccPolicy : public Policy
{
public:
  LbccPolicy();

public:
  static const std::string POLICY_NAME;

private:
  virtual void
  doAfterInsert(iterator i) override;

  virtual void
  doAfterRefresh(iterator i) override;

  virtual void
  doBeforeErase(iterator i) override;

  virtual void
  doBeforeUse(iterator i) override;

  virtual void
  evictEntries() override;

private:
  /** \brief moves an entry to the end of queue
   */
  void
  insertToQueue(iterator i, bool isNewEntry);

  iterator
  calculateRequestTrend(Queue m_queue, Ptr<Item> item);

  iterator
  calculateTotalTrend(Queue m_queue);

  iterator
  calculateRequestRate(Queue m_queue,Ptr<Item> item);

  iterator
  calculatePopularity(Queue m_queue, Ptr<Item> item);

  iterator
  calculateContentStorePosition(Ptr<Item> item);


private:
  Queue m_queue;
};

} // namespace lru

using lbcc::LbccPolicy;
```

```

} // namespace cs
} // namespace nfd

#endif // NFD_DAEMON_TABLE_CS_POLICY_LBCC_HPP
```

B.5 LBCC Simulation File

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/ndnSIM-module.h"

namespace ns3 {
int
main(int argc, char* argv[])
{
  CommandLine cmd;
  cmd.Parse(argc, argv);

  AnnotatedTopologyReader topologyReader("", 1);
  topologyReader.SetFileName("src/ndnSIM/examples/topologies/topo-tree-62
  topologyReader.Read();

  // Install NDN stack on all nodes
  ndn::StackHelper ndnHelper;
  //ndnHelper.SetOldContentStore("ns3::ndn::cs::Lru", "MaxSize","100");
  ndnHelper.SetOldContentStore("ns3::ndn::cs::lbcc", "MaxSize","1000"); /
  ndnHelper.InstallAll();

  // Choosing forwarding strategy
  ndn::StrategyChoiceHelper::InstallAll("/prefix", "/localhost/nfd/strate

  // Installing global routing interface on all nodes
  ndn::GlobalRoutingHelper ndnGlobalRoutingHelper;
  ndnGlobalRoutingHelper.InstallAll();

  // Getting containers for the consumer/producer
  Ptr<Node> consumers[32] = {
    Names::Find<Node>("Src1"),
    Names::Find<Node>("Src2"),
    Names::Find<Node>("Src3"),
    Names::Find<Node>("Src4"),
    Names::Find<Node>("Src5"),
    Names::Find<Node>("Src6"),
    Names::Find<Node>("Src7"),
    Names::Find<Node>("Src8"),
    Names::Find<Node>("Src9"),
    Names::Find<Node>("Src10"),
    Names::Find<Node>("Src11"),
    Names::Find<Node>("Src12"),
    Names::Find<Node>("Src13"),
    Names::Find<Node>("Src14"),
    Names::Find<Node>("Src15"),
    Names::Find<Node>("Src16"),
    Names::Find<Node>("Src17"),
    Names::Find<Node>("Src18"),
    Names::Find<Node>("Src19"),
    Names::Find<Node>("Src20"),
    Names::Find<Node>("Src21"),
    Names::Find<Node>("Src22"),
    Names::Find<Node>("Src23"),
    Names::Find<Node>("Src24"),
    Names::Find<Node>("Src25"),
    Names::Find<Node>("Src26"),
    Names::Find<Node>("Src27"),
    Names::Find<Node>("Src28"),
    Names::Find<Node>("Src29"),
    Names::Find<Node>("Src30"),
    Names::Find<Node>("Src31"),
    Names::Find<Node>("Src32")
};

  Ptr<Node> producer1 = Names::Find<Node>("Root1");
  Ptr<Node> producer2 = Names::Find<Node>("Root2");
  Ptr<Node> producer3 = Names::Find<Node>("Root3");
  Ptr<Node> producer4 = Names::Find<Node>("Root4");
```

```
Ptr<Node> producer5 = Names::Find<Node>("Root5");
Ptr<Node> producer6 = Names::Find<Node>("Root6");

for (int i = 0; i < 32; i++) {
  ndn::AppHelper consumerHelper("ns3::ndn::ConsumerCbr");
  consumerHelper.SetAttribute("Frequency", StringValue("200")); // 200

  // Each consumer will express the same data /root/<seq-no>
  consumerHelper.SetPrefix("/Root");
  ApplicationContainer app = consumerHelper.Install(consumers[i]);
  app.Start(Seconds(0.01 * i));
}

ndn::AppHelper producerHelper("ns3::ndn::Producer");
producerHelper.SetAttribute("PayloadSize", StringValue("1024"));


ndnGlobalRoutingHelper.AddOrigins("/Root1", producer1);
producerHelper.SetPrefix("/Root1");
producerHelper.Install(producer1);

ndnGlobalRoutingHelper.AddOrigins("/Root2", producer2);
producerHelper.SetPrefix("/Root2");
producerHelper.Install(producer2);


ndnGlobalRoutingHelper.AddOrigins("/Root3", producer3);
producerHelper.SetPrefix("/Root3");
producerHelper.Install(producer3);


ndnGlobalRoutingHelper.AddOrigins("/Root4", producer4);
producerHelper.SetPrefix("/Root4");
producerHelper.Install(producer4);

ndnGlobalRoutingHelper.AddOrigins("/Root5", producer5);
producerHelper.SetPrefix("/Root5");
producerHelper.Install(producer5);
```

```


  ndnGlobalRoutingHelper.AddOrigins("/Root6", producer6);
  producerHelper.SetPrefix("/Root6");
  producerHelper.Install(producer6);

  // Calculate and install FIBs
  ndn::GlobalRoutingHelper::CalculateRoutes();

  Simulator::Stop(Seconds(20.0));

  ndn::CsTracer::InstallAll("cs-popularity-based-cache.txt", Seconds(1));

  Simulator::Run();
  Simulator::Destroy();

  return 0;
}

} // namespace ns3

int
main(int argc, char* argv[])
{
  return ns3::main(argc, argv);
}
```

Generated using Postgraduate Thesis LaTeX Template, Version 0.97. Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.

This thesis was generated on Monday 3rd February, 2020 at 4:32am.